Online Trace Reordering for Efficient Representation of Event Partial Orders

by

Muhammad Bilal Sheikh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Muhammad Bilal Sheikh

Abstract

Distributed and parallel applications not only have distributed state but are often inherently non-deterministic, making them significantly more challenging to monitor and debug. Additionally, a significant challenge when working with distributed and parallel applications has to do with the fundamental requirement of determining the order in which certain actions are performed by the application. A naive approach for ordering actions would be to impose a single order on all actions, i.e., given any two actions or events, one must happen before the other. A global order, however, is often misleading, e.g., two events in two different processes may be causally independent yet one may have occurred before the other. A partial order of events, therefore, serves as the fundamental data structure for ordering events in distributed and parallel applications.

Traditionally, Fidge/Mattern timestamps have been used for representing event partial orders. The size of the vector timestamp depends on the number of parallel entities (traces) in the application, e.g., processes or threads. A major limitation of Fidge/Mattern timestamps is that the total size of timestamps does not scale for large systems with hundreds or thousands of traces. Taylor proposed an efficient offset-based scheme for representing large event partial orders by representing deltas between timestamps of successive events. The offset-based schemes have been shown to be significantly more space efficient when traces that communicate the most are close to each other for generating the deltas (offsets). In Taylor's offset-based schemes the optimal order of traces is computed offline. In this work we adapt the offset-based schemes to dynamically reorder traces and demonstrate that very efficient scalable representations of event partial orders can be generated in an online setting, requiring as few as 100 bytes/event for storing partial order event data for applications with around 1000 processes.

Acknowledgements

I would like to thank my supervisor Prof. David Taylor for his extensive support, valuable guidance and patience throughout my Masters. I would also like to thank Prof. Paul Ward and Prof. Bernard Wong for being my thesis readers. I would further like to thank my Shoshin lab mates for all the valuable discussions and feedback. Finally, I would like to thank all my friends in Waterloo and back home for all the support and good times.

Dedication

I dedicate this work to my parents.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A distributed system is a collection of decoupled components that appears to an end user as a single coherent system. The various components in a distributed system run autonomously and coordinate their activities by passing messages to each other over a network. Distributed systems and applications offer a number of advantages over stand-alone applications including, availability, performance and incremental scalability. These systems however, are significantly more complex than stand-alone systems. Since many different components are involved, there can be several sources of problems in a distributed system, from hardware and network failures, to software bugs, data corruption and system overload [45]. Therefore, visibility into the workings of these systems is essential for performance and failure analysis, improving resource utilization, and debugging.

1.1 Prevalence of Distributed Applications

Many of today's widely used computing applications are distributed in nature. These applications are highly complex and run on heterogeneous hardware. Furthermore, these applications operate at an unprecedented scale, running on hundreds and thousands of machines. Today, it is significantly more cost-effective to build and operate these large distributed applications. Reduction in cost is driven by ever lowering costs of computing (Moore's Law), more reliable network infrastructures, and widespread adoption of shared-nothing architectures. Additionally, virtualization of physical resources, emergence of *-as-a-service from storage-as-a-service to infrastructure-as-a-service [2, 4, 3, 6, 7], and the availability of highly distributed application frameworks [5, 13, 15, 30, 58, 63] has further lowered the barrier to entry for the development of even larger distributed applications.

Multi-processor systems are the norm these days, as raising the clock speed of individual processors is becoming impractical from a hardware perspective because of several physical issues including too much heat dissipation, too much power consumption, and current leakage problems [28]. Sequential programs are no longer sufficient and require fundamental changes to extract performance from these multi-processor systems. Therefore, to take advantage of multi-processor systems, the research and development community has turned their attention toward the development of parallel applications [11, 23]. This has in turn led to more focus on multi-threaded applications and a significant shift towards functional programming languages [10, 50, 35], which offer a more natural paradigm for writing parallel applications.

These distributed and parallel applications are inherently more complex than stand-alone sequential applications. A web search for example, touches thousands of machines and more than a few dozen separate services [21]. Small software bugs in these applications

can cause massive failures, affecting hundreds of thousands of users [48]. Similarly, as individuals and businesses become increasingly dependent on various distributed services, the consequences of service failures can be very significant. A recent example of this is the outage of Amazon's elastic compute cloud (EC2) [2] caused by a number of small software and hardware failures [1]. The outage resulted in thousands of websites and services becoming inaccessible to millions of users. These trends towards always available large distributed services and the potential consequences of bugs and failures, therefore provide an even greater impetus for improving the tools used for monitoring and debugging these applications.

1.2 Monitoring Distributed and Parallel Applications

The purpose of monitoring applications is varied and includes debugging [12, 20], performance and failure analysis [14, 26, 31, 57], capacity planning [37], and tuning and control [32, 41]. Monitoring involves collection, measurement, and processing of data emitted by an application during execution [42]. Typically, the application under observation is instrumented to generate events when specific actions are performed. The events generated by the monitored application are transmitted to a separate monitoring entity which processes and stores these events for various monitoring and debugging purposes. Generally, more collected data can give greater insights into the workings of the application under observation, however, care should be taken when instrumenting an application for data collection. Too much instrumentation can produce copious amounts of data that can easily overwhelm the monitoring entity. Furthermore, the level of instrumentation has a direct impact on the actual runtime behavior of the monitored application.

Monitoring and debugging distributed applications is even harder as these applications

present some unique challenges [24]:

1. Distributed and parallel applications are inherently non-deterministic. Consider, for example, a number of threads running on a system. Although each thread will execute its steps in a predictable order, the overall execution of the threads would be interleaved. As a result we could get a different execution history each time the application is run.

2. It is often impractical to have a global clock in distributed applications. Each system has a local clock; however, since these clocks are not synchronized, one cannot determine the global order of execution. Even if a global order is imposed, it is misleading and cannot be used for visualization, monitoring and debugging these applications. This presents another significant challenge when monitoring these applications.

3. Distributed systems often have distributed state and different components communicate with each other using message passing. Additionally, many parallel applications use a message-passing concurrency model as opposed to a shared memory concurrency model. The absence of centralized state is yet another challenge when monitoring and debugging these applications.

4. A fourth and a significant challenge in working with distributed applications is that the execution of these applications can produce huge amounts of event data. This can easily overwhelm a monitoring tool that is trying to process that data for various debugging and monitoring tasks.

1.3 Motivation

Monitoring and finding faults quickly in today's always available distributed services to prevent failures is increasingly critical. Therefore, though offline debugging and analysis is quite useful, there is a growing need for real-time monitoring and fault analysis of distributed and parallel applications.

In practice, the event data generated by distributed applications is stored in large event log files for later use. This approach generally works well for offline monitoring tasks like execution replay [18, 55] and event-pattern search [22, 49], that either require the complete event data or are computationally expensive. However, due to the copious amounts of event data generated by distributed applications, monitoring tools face severe scalability issues when processing data in real-time. This in turn can greatly limit the capabilities of these tools, and can force system administrators to run even the simplest of these operations in offline mode.

In order to overcome these scalability issues, there is a need for efficient representation of event data, which can significantly reduce the space required for storing the event data. Such representation is not only beneficial for existing offline and online algorithms for visualization [62], replay, and search by making them less I/O bound, but can further help in the development of cleverer algorithms, potentially making some currently prohibitive debugging operations possible in real-time. In this work we focus on such an efficient representation of event data, i.e., efficient representation of event partial orders [61] (detailed in Chapter 4). More specifically we develop a number of trace-reordering schemes for generating efficient representations of event data in an online manner (Chapter 5).

Over the years, a number of approaches have been developed for efficiently representing event data, however, as we discuss in Chapter 3, most of these techniques are either not

efficient enough or have too high an access cost if they succeed in sufficiently reducing the size of event data. Furthermore, a major limitation of these techniques is that they cannot be used for generating efficient representations in an online setting.

1.4 Contributions

As described in Section 1.2, a global order of execution, if it can be determined, is of little value when monitoring and debugging distributed and parallel applications. A consequence of this limitation is that we need to work with partial order of events that can be constructed based on the events generated by a target application.

Traditionally, the partial-order relation on the ordering of events is represented using Fidge/Mattern timstamps [24, 44]. A limitation of using Fidge/Mattern timestamps is that the size of the partial-order representation grows linearly with the number of parallel entities, e.g., processes, so the space required for representation grows proportional to the product of the number of events and the number of entities. Therefore it does not scale for large applications.

Taylor [61] proposed an offset-based event-partial-order representation that scales for applications with a large number of processes. He further showed that the offset-based schemes are most efficient when the different parallel entities (referred to as *traces*) in the application are ordered based on the level of communication with other traces, i.e., traces that communicate heavily are adjacent to each other. The more efficient variant of the offset-based schemes that utilizes communication-based trace orders, generates this trace order after seeing all the events in the application.

In this work we adapt the offset-based partial-order-representation scheme to work with

a dynamic trace-reordering scheme. Our proposed scheme is directly built into the C++ variant of the Partial Order Event Tracer (POET) [42], a tool for monitoring and debugging distributed and parallel applications. We further adapt POET to store and provide rapid access to the event partial order for various monitoring and debugging facilities. Some significant contributions of our work are as follows:

1. We adapt the offset-based partial order representation schemes proposed by Taylor [61] to work in an online manner by periodically reordering traces. We propose a dynamic application-independent scheme for ordering traces online to facilitate construction of real-time scalable event partial orders that can be used for monitoring and debugging.

2. We explore a number of different policies to keep the overhead of the online representation scheme as low as possible by limiting the number of times the traces are reordered without compromising the space effectiveness of the offline offset-based schemes.

3. We propose a layered client architecture for POET [42] for developing different monitoring and debugging facilities and build the online offset-based representation client using the proposed layered architecture.

4. Lastly, we evaluate the space efficiency achieved by our online extensions to the offline offset-based representation algorithms proposed by Taylor [61].

1.5 Organization

This thesis is organized as follows: Chapter 2 offers an overview of the ordering of events in distributed and parallel applications. It describes the traditional approaches for dealing with event orders, the size and scalability of these approaches, and the unique challenges associated with monitoring distributed and parallel applications. Chapter 3 summarizes the key aspects of an efficient partial order representation and reviews the existing work on efficiently representing partial orders both in distributed systems and in database community. Chapter 4 details the offset-based representation schemes that form the basis of our work. In Chapter 5 we propose and adapt the offset-based schemes to an online setting, then evaluate and analyze the space and computational efficiency of the online extensions to the offset-based partial order representations. Lastly, in Chapter 6 we conclude with closing remarks and identify areas of future work.

Chapter 2

Representing Event Orders

2.1 Introduction

For monitoring and debugging purposes, applications are instrumented to emit events when certain actions are performed. A fundamental requirement of any monitoring and debugging utility is to know the order in which various actions are performed by the application. More specifically, given two events (for two actions), we need to determine if one event happened before the other. In a sequential process, local or physical clocks are sufficient to determine the ordering of events. Consider for example Figure 2.1a, representing a single sequential entity P_1. Event a occurs at time T_a and event b occurs at time T_b as measured by the local clock C_1. In this scenario, given T_a and T_b we can determine the order in which events a and b occurred in P_1. If $T_a < T_b$ then event a happened before event b, alternatively if $T_a > T_b$ then event b happened before event a. By similar comparison, we can determine the complete order of events emitted during the execution of P_1 and can construct a timeline for P_1 as shown in Figure 2.1b.

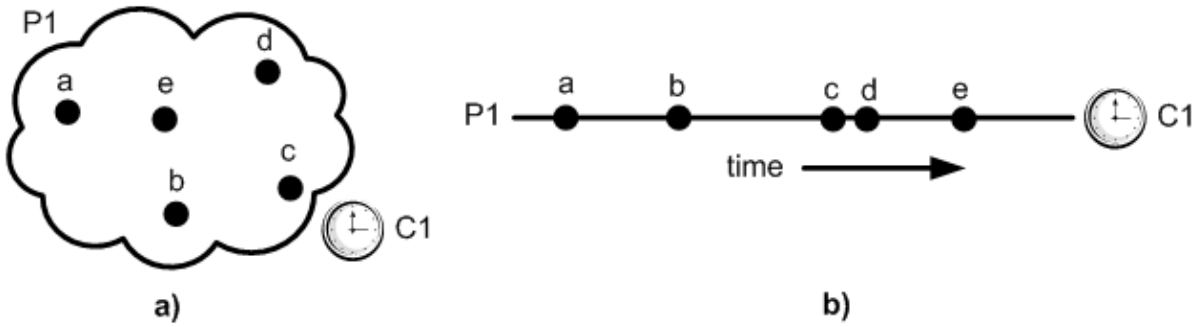

Figure 2.1: Timeline representation of events emitted by a single-process application

2.2 Ordering Events in Distributed Applications

In a distributed application, however, it is often impractical to have a single global clock or equivalently, to completely synchronize local clocks. Figure 2.2a shows the events generated in a hypothetical distributed application with two processes P_1 and P_2. C_1 and C_2 are the unsynchronized local clocks for P_1 and P_2. The corresponding event timelines for the processes P_1 and P_2 are shown in Figure 2.2b. Following the above discussion for a single process, we know that event b occurs before event c. Additionally, we can further conclude that event d on process P_1 happens before event e on process P_2, simply because an event generated when a message is sent (*send event*) must causally precede an event generated when that same message is received (*receive event*). This precedence relation is used to represent causality in a distributed application, and more formally we can say that $a \rightarrow b$ and $c \rightarrow d$. The precedence relation ($\rightarrow$) has the following definition:

Definition 1 (Precedence Relation) *The precedence relation ($\rightarrow$) has the following three properties:*

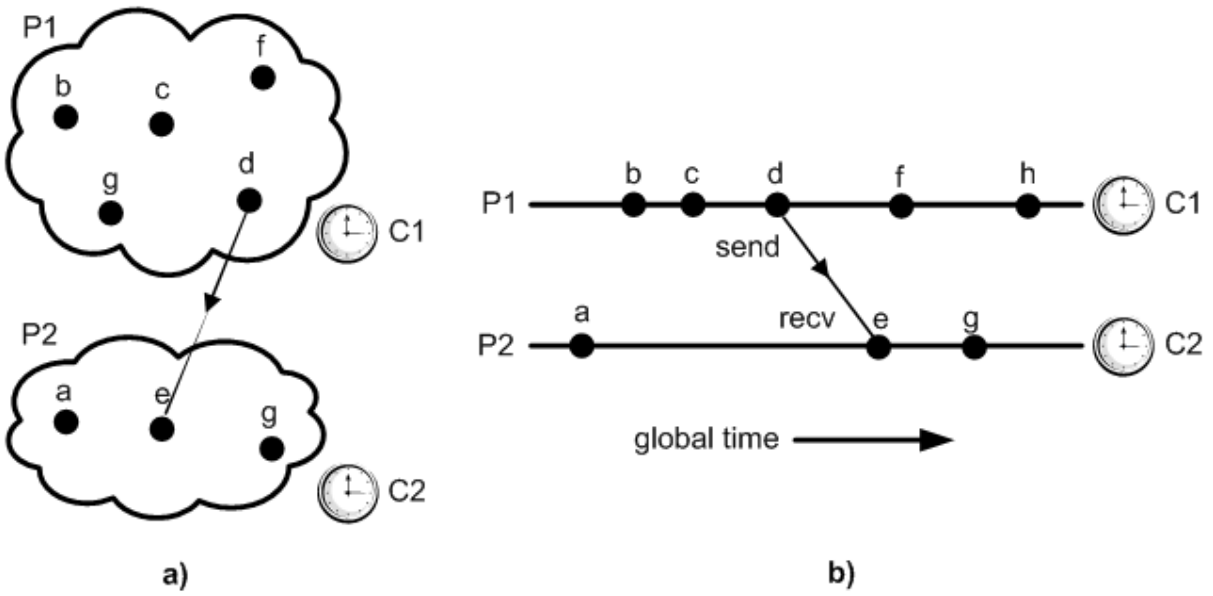

Figure 2.2: Events emitted by two processes in a distributed application

1. *Irreflexive:* $a \nrightarrow a$

2. *Transitive: If* $a \rightarrow b$ *and* $b \rightarrow c$ *then* $a \rightarrow c$

3. *Anti-Symmetric: If* $a \rightarrow b$ *then* $b \nrightarrow a$

Continuing with our example, assume that events e and f have timestamps T_e and T_f as assigned by local clocks C_2 and C_1. If we further assume that $T_e < T_f$ and the difference between the two timestamps is given by T_{f-e} $(= T_f - T_e)$, then event $e \rightarrow f$ only if $C_1 - C_2 < T_{f-e}$. Without this information about the synchrony of clocks C_1 and C_2, we cannot determine the precedence relation between e and f, i.e. if $e \rightarrow f$ or if $f \rightarrow e$. Therefore, for our purposes event e and event f are causally independent or concurrent ($\|$) irrespective of the actual physical time at which these events occurred. Given Definition 1 for precedence, two events are concurrent if neither precedes the other (Definition 2).

Definition 2 (Concurrency) *$a \parallel b$ if and only if $a \nrightarrow b$ and $b \nrightarrow a$*

The above example demonstrates that without synchronized local clocks we cannot determine the precedence relation between all events in a distributed application. Therefore, instead of using physical clocks for completely ordering events in a distributed application, which can be inaccurate and misleading, it is desirable to work with the partial order determined by the precedence relation ($\rightarrow$). A partial order is a relation on a set that is *reflexive*, *transitive*, and *anti-symmetric*. The precedence relation ($\rightarrow$) as defined above is a partial-order relation on the set of events, making the set of events in a distributed application a *partially ordered set* (POSET).

Definition 3 (Partially Ordered Set) *A partially ordered set (or poset, or partial order) is a pair (X, P) where X is a finite set and P is a reflexive, anti-symmetric, and transitive binary relation on X.*

In fact since the precedence relation is irreflexive, it forms a *strict partial order* and throughout our discussion we will assume that we are dealing with a strict partial order.

2.3 Representing Event Partial Orders

2.3.1 Transitive Closure and Reduction of Partial Order

The partial-order relation can be represented as a directed acyclic graph (DAG) using *reachability*. A vertex in a directed graph is reachable from another vertex if there exists a path between the two vertices. More formally, the reachability relation has the following definition:

Definition 4 (Reachability) *For a directed graph $D = (V, A)$, the reachability relation of D is the transitive closure of its arc set A, which is to say the set of all ordered pairs (s, t) of vertices in V for which there exist vertices $v_0 = s, v_1, \ldots, v_d = t$ such that (v_{i-1}, v_i) is in A for all $1 \leq i \leq d$.*

Figure 2.3a shows the DAG for the partial-order relation on the set of events in our hypothetical distributed application. In fact, the DAG represents the *transitive closure* of the partial-order relation ($\rightarrow$) on the events. The definition of transitive closure is given as follows:

Definition 5 (Transitive Closure) *The transitive closure of a binary relation R on a set X is the minimal transitive relation R' on X that contains R. Thus $aR'b$ for any elements a and b of X provided that there exists $c_0, c_1, ..., c_n$ with $c_0 = a, c_n = b$ and $c_{r-1}Rc_r$ for all $1 \leq r \leq n$.*

An edge in the DAG represents precedence ($\rightarrow$) between two events. For example, by looking at Figure 2.3a we can conclude that event b precedes event g. The transitive closure of a partial order can be represented using a *connectivity matrix* and can be constructed from an *adjacency matrix* using Warshall's algorithm [72]. The computational complexity of constructing the connectivity matrix is $O(E^3)$ where E is the number of events emitted by our instrumented application. Once we have the connectivity matrix, the complexity of determining precedence between two events is $O(1)$, however, note that the space complexity for representing the convex closure using a connectivity matrix is $O(E^2)$. For an application that generates 100000 events during execution, the connectivity matrix alone would require approximately 10GB of space, making the transitive-closure representation of event partial orders infeasible for most real applications.

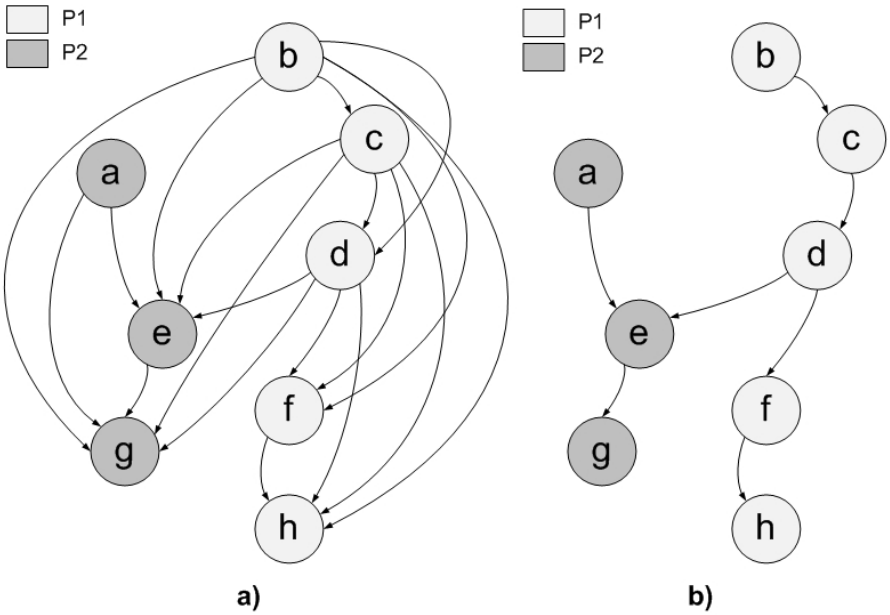

Figure 2.3: DAGs of a) transitive closure and b) transitive reduction of partial order

To reduce the space needed for representing event partial orders, we can take advantage of a specific property of DAGs, i.e., any two DAGs with the same reachability relation represent the same partial order. In fact, a DAG representing the transitive closure of a partial order has the maximum number of edges of all such DAG representations of the same partial order. Therefore to save space, an alternative is to represent the partial order by its *transitive reduction*. A DAG representing the transitive reduction of a partial order uses the least number of edges and has the same reachability as the DAG representing the transitive closure of the partial order. Transitive reduction is defined as follows:

Definition 6 (Transitive Reduction) *A transitive reduction of a binary relation R on a set X is a minimal relation R' on X such that the transitive closure of R' is the same as the transitive closure of R.*

Note that the transitive-reduction representation sets the lower bound on the space required for representing a partial-order relation. Figure 2.3b shows the DAG representing the transitive reduction of the partial-order relation on the set of events in our example. The space complexity in this case is $O(E + M)$ where E is the number of events and M is the number of messages exchanged between processes. A major drawback of representing an event partial order by its transitive reduction is the cost of determining the precedence relation between two events. The computational complexity using a depth-first approach is $O(E + M)$ which is too high for most debugging and monitoring facilities that need to carry out large numbers of precedence-testing operations in real time.

2.3.2 Lamport Clocks

An alternative to using transitive closure or transitive reduction of event partial orders is to attach a number to each event by maintaining a logical clock for each process in the distributed application. These logical clocks were first introduced by Lamport [43]. More precisely, a clock C_i is defined for each process in the application and each event a on that process is assigned a value $C_i(a)$. Note that these clocks have no assumption about the actual time and therefore do not depend on the physical clocks. The clock value for each event is generated using the following two rules [43]:

1. *Rule 1:* Each process P_i increments C_i between any two successive events.

2. *Rule 2:* (a) If event a is the sending of a message m by process P_i, then the message m contains a timestamp $T_m = C_i(a)$. (b) Upon receiving a message m, process P_i sets C_i greater than or equal to its present value and greater than T_m.

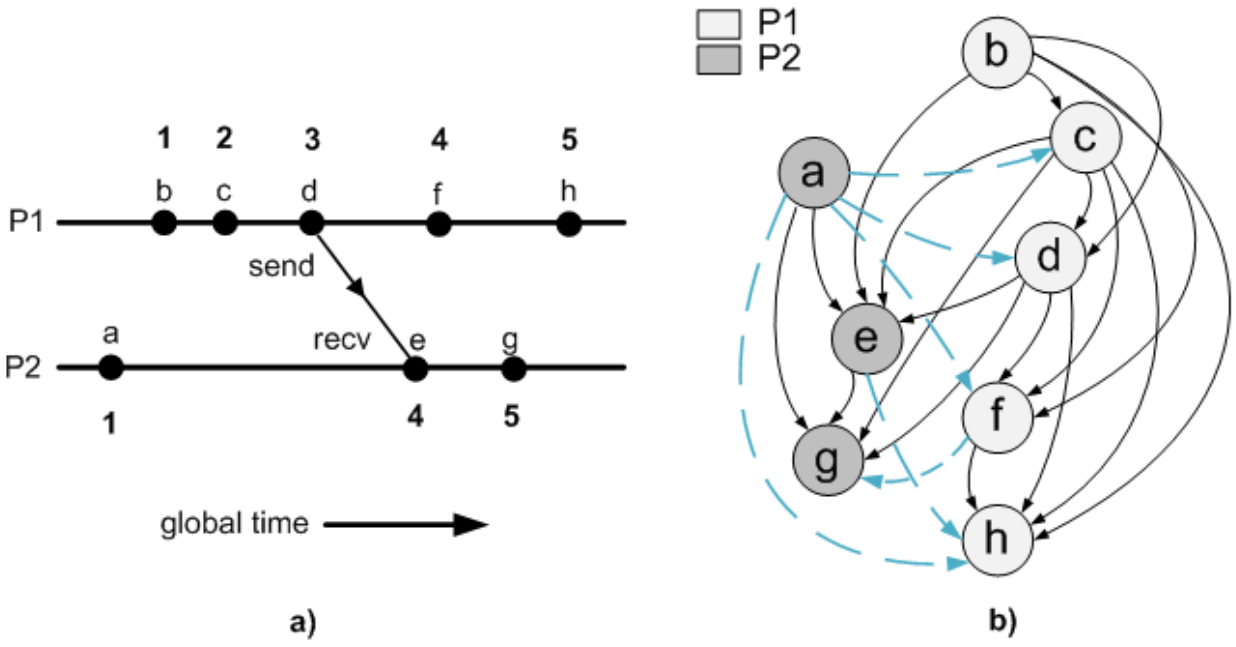

Figure 2.4: Ordering events using Lamport clocks

In our example, using smallest positive integers for clocks we get $C_2(a) = 1$, $C_1(b) = 1$, $C_1(c) = 2$ and $C_1(d) = 3$ for events a,b,c and d by applying rule 1 for each event. Following rule 2a, when P_1 sends the message m to P_2, the timestamp $T_m = 3$ is sent along the message to P_2. When P_2 receives the message m with timestamp T_m, rule 2b is used by the local clock C_2 to generate timestamp $C_2(e) = max(T_m + 1, C_2(a)) = 4$. Similarly, events f, g and h are assigned their respective clock values as shown in Figure 2.4a. A limitation of using Lamport logical clocks is that these clocks impose a total order on events where none exists. In terms of a DAG representation (Figure 2.4b), using Lamport clocks results in the addition of edges (dashed lines) that are not present in the transitive-closure representation of the partial order. For example, $C_1(h) = 5 > C_2(e) = 4$, but, as discussed

in Section 2.2, e does not happen before h, i.e., $e \nrightarrow h$. The DAG in Figure 2.4b, therefore, does not have the same reachability relation as the ones representing the partial order in Figure 2.3. Since Lamport clocks impose a total order on events and therefore cannot be used for determining precedence relations, we won't consider this method any further in our work.

2.3.3 Fidge/Mattern Vector Timestamps

A vector-timestamping approach associates a vector of clock values with each event in a distributed application. These vector timestamps can then be compared to determine the precedence relation between events. Over the years a number of vector timestamping schemes have been proposed that preserve the partial-order relation on events, including Fidge/Mattern [24, 44], Fowler/Zwaenepoel [25], Jard/Jourdan [36], Ore [51], Summer's cluster-timestamps [59], and Ward's dimension-bound [68] and centralized-cluster timestamps [66, 71]. These timestamps add a number of edges to the transitive reduction of the partial order and differ from each other in how they are generated, the space required for timestamps, and consequently the computation cost of testing precedence. Among these timestamps, Fidge/Mattern timestamps have found widespread applicability, mainly because of the simplicity of creating timestamps for new events in real time and, more importantly, because a single comparison is required for precedence testing. For our work, we focus on adapting the efficient partial-order representation schemes presented in [61] to an online setting. These schemes make use of Fidge/Mattern timestamps for representing event partial orders. In Chapter 3 we discuss some of these timestamping algorithms in the context of existing schemes for conserving space when representing event partial orders, however, a reader looking for a detailed comparison of these timestamps is directed

to Ward's work [69]. We next describe the algorithm for ordering events in a distributed application by assigning a Fidge/Mattern timestamp to each event.

Let P_1, P_2, ... P_N be each of the N traces in a distributed application. Each trace P_i maintains a vector clock T_i of size N, which is used for assigning timestamps to events on P_i. The following rules, as describe by Fidge and Mattern, are followed:

1. Initialize the N-element vector clock for each trace P_i to 0, i.e.,

   $T_i(k) = 0, i = 1 \ldots N, k = 1 \ldots N.$

2. For each event a occurring on trace P_i, update T_i by incrementing the ith element of T_i by 1. Assign the updated T_i to event a. Specifically,

   $T_i[i] = T_i[i] + 1$

   $T_a = T_i$

3. For a send of message m_{ij} from trace P_i to P_j, (a) update T_i and assign the updated timestamp to the send event a_s on P_i according to rule 2. (b) Send the updated T_i to trace P_j along with the message m_{ij}.

4. For a message received on trace P_j and sent from trace P_i (m_{ij}) with an attached timestamp T_m, take the following steps:

   (a) update trace P_j's local timestamp T_j as follows:

   $T_m[i] = T_m[i] + 1$

   $T_j[j] = T_j[j] + 1$

   $T_j[k] = max(T_m[k], T_j[k]), k = 1 \ldots N$

   (b) assign the updated timestamp T_j to the receive event a_r, i.e.,

   $T_{a_r} = T_j$

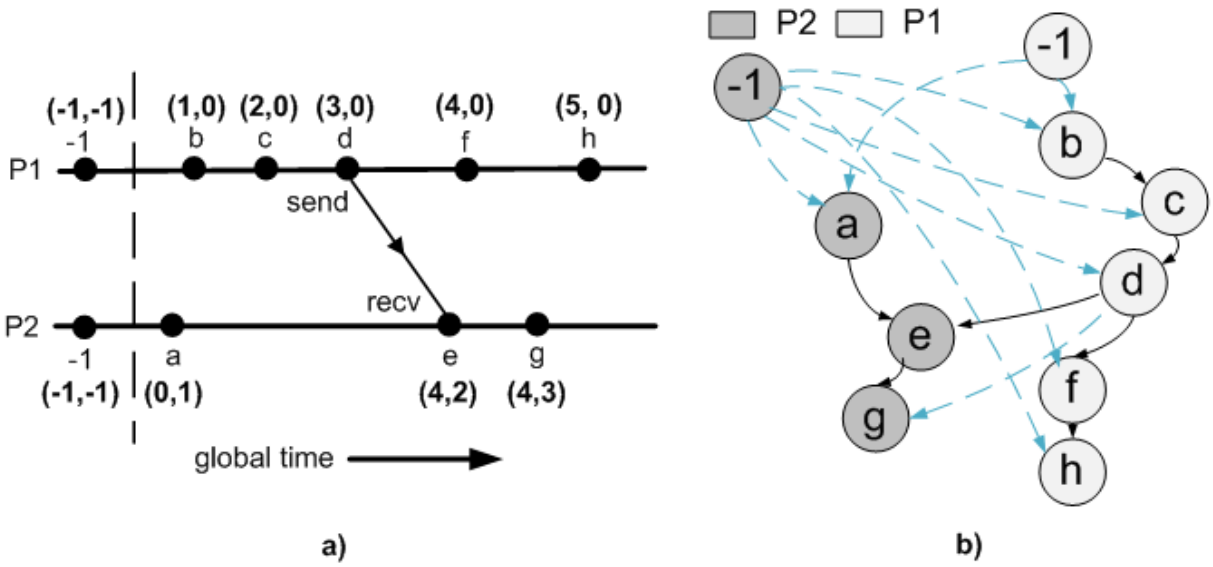

Figure 2.5: Fidge/Mattern timestamps

In our example, both processes P_1 and P_2 will have local clocks T_1 and T_2 initially set to 0, i.e., $T_1 = T_2 = (0, 0)$ (rule 1). Events a, b and c will have timestamps $T_a = (0, 1)$, $T_b = (1, 0)$ and $T_c = (2, 0)$ following the application of rule 2 for each of the events. For the message sent between P_1 and P_2, event d will have timestamp $T_d = (3, 0)$ following rule 3a and a copy of the timestamp will be sent as T_m to trace P_2. On receiving the message on trace P_2, the local timestamp T_j is updated to $(4, 2)$ and a copy of the timestamp is assigned to event e according to rule 4. Lastly, events f and h are assigned timestamps by incrementing the local timestamp T_1 and event g is assigned a timestamp by incrementing timestamp T_2. Figure 2.5a shows the Fidge/Mattern timestamps for each event in our application. Precedence and concurrency between two events that are timestamped using the Fidge/Mattern algorithm can be determined as follows:

Theorem 1 (Precedence) *Let a and b be two events on traces P_i and P_j with timestamps T_a and T_b then $a \rightarrow b$ if and only if $T_a[i] < T_b[i]$.*

Theorem 2 (Concurrency) *Let a and b be two events on traces P_i and P_j with time-stamps T_a and T_b then $a \parallel b$ if and only if $T_a[i] \not< T_b[i]$ and $T_b[j] \not< T_a[j]$.*

The Fidge/Mattern algorithm benefits from the notion of traces in a distributed application, allowing for the determination of causality between two events in constant time. This constant-time precedence testing is facilitated by adding a number of edges to the transitive reduction of the partial order. Specifically, each event a has N incoming edges from the greatest event on each of the N traces that causally precedes a. Such an event on each trace is referred to as the *greatest predecessor* of a on that trace, given formally:

Definition 7 (Greatest Predecessor) *The greatest predecessor of an event, a, on trace P_i denoted $GP_{P_i}(a)$ is the single-element set containing the most-recent event, $\{e\}$, on trace P_i that happens before a i.e. $e \rightarrow a$, or the empty set, $\{\}$, if no such event exists.*

Figure 2.5b shows the DAG representing Fidge/Mattern timestamps for our example. The edges extra to the transitive reduction are shown as dashed edges. Since, local time-stamps T_1 and T_2 for each process are initialized to 0 (rule 1) we introduce a hypothetical "-1" event on each trace P_i. These -1 events on each trace P_i act as the initial greatest predecessors for the actual events on each trace P_j until trace P_j receives a message m_{ij} from trace P_i. Note that these -1 events are added only to show the edges added when Fidge/Mattern timestamps are used and do not exist in practice. In Figure 2.5 for example, the greatest predecessors of event a on each trace are the -1 events. Similarly, the greatest predecessor for event g is d on trace P_1 and e on trace P_2.

So far, our discussion assumes that all communication in a distributed application is asynchronous, i.e., after sending a message, a trace does not wait for the reply and continues to execute, generating new events. In Figure 2.5a, event d is an asynchronous send and

e is the corresponding asynchronous receive. Not all communication is asynchronous and, alternatively, a process can block after sending a message until the message is received. To handle synchronous communication, Cheung [17] introduced the following extension to the Fidge/Mattern algorithm:

1. (a) Let m_{ij} be a synchronous message sent from trace P_i to trace P_j, with a and b the send and receive events on each trace. The following steps are taken for assigning new timestamps:

   $T_i[i] = T_i[i] + 1$

   $T_j[j] = T_j[j] + 1$

   $T_i[k] = T_j[k] = max(T_i[k], T_j[k]), k = 1 \ldots N$

   $T_a = T_b = T_i$

2. (b) In preparation for the next events on each trace, update the local clocks for P_i and P_j as follows:

   $T_i[j] = T_i[j] + 1$

   $T_j[i] = T_j[i] + 1$

Figure 2.6 shows an extended version of the example event timeline, with synchronous communication. Event i is a synchronous send and event j is a synchronous receive event. Note that events i and j have the same timestamp. Furthermore, the local timestamp T_i is updated to $(6, 5)$ and T_j is updated to $(7, 4)$ before events k and l are assigned timestamps.

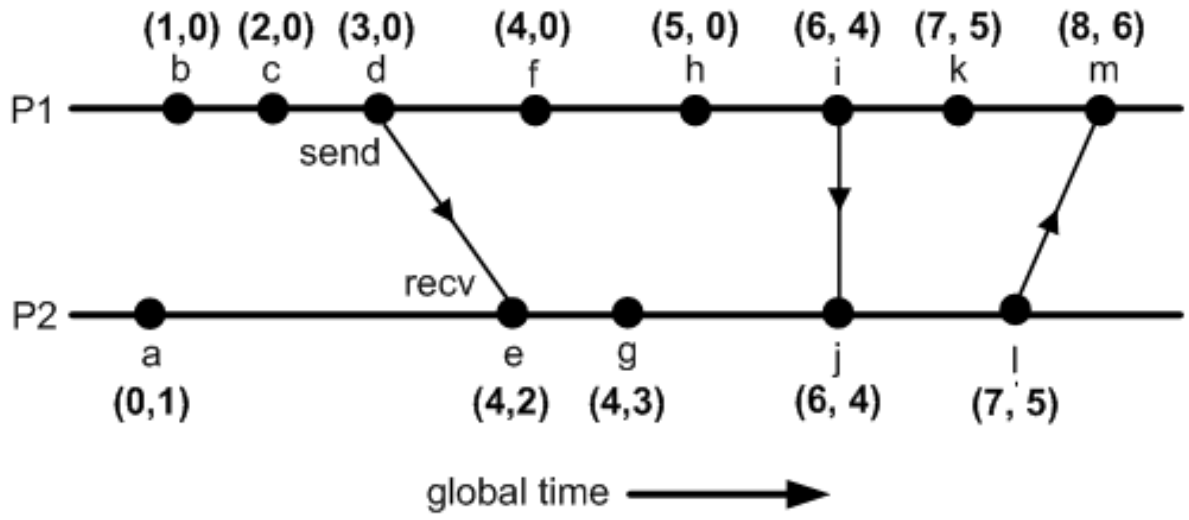

Figure 2.6: Fidge/Mattern timestamps with synchronous communication

2.4 Case for Efficient Representation of Event Orders

2.4.1 Size of Representations

A DAG of the transitive closure of a partial order can be represented using a connectivity matrix with a space complexity of $O(E^2)$ where E is the total number of events. On the other hand, in a DAG of the transitive reduction of the partial order, for each event on trace P_i there is an incoming edge from the greatest predecessor on that trace and all receive events have an additional incoming edge from a send event. This results in a space complexity of $O(E + M)$ where E is the number of events and M is the number of messages. Although the space required is small, the cost of determining precedence is too high when working with the transitive reduction of a partial order, therefore, as stated in Section 2.3.1, a standard approach is to use Fidge/Mattern timestamps.

By taking advantage of traces in a distributed application, Fidge/Mattern timesamps reduce the computational complexity of determining precedence to $O(1)$, which is the same as for the transtive closure of a partial order. In a DAG representation of Fidge/Mattern timestamps, each event has N incoming edges from the greatest predecessors on each of the N traces in a distributed application. The space complexity therefore is directly proportional to the number of events E and the number of traces N, i.e. $O(N \times E)$. This is an improvement over the transitive-closure space requirement of $O(E^2)$ as the number of processes N is significantly less than the number of events E (i.e., $N << E$) for most real applications.

For distributed applications with large numbers of processes, the space required for storing Fidge/Mattern timestamps can still increase quickly, as it directly depends on the number of processes. Consider for example a distributed application with 1000 traces with an average of 1000 events on each trace, resulting in a total of 1000000 events. Each event will be represented by a 1000-integer vector timestamp. Assuming 4 bytes for an integer, the size of each timestamp will be $1000 \times 4\text{B} \approx 4\text{KB}$, making the total space needed for storing Fidge/Mattern timestamps for all events approximately 4GB. Note that this is only the size of the vector timestamps for representing the partial order and does not include other typical information associated with an event like the event number, trace number, event type (e.g. send over socket), and some textual information.

The always available distributed systems in use today can generate massive event logs and efficiently storing and managing these logs for later processing is an active area of research [9]. As noted above, these event logs alone do not store the partial-order representation required for monitoring and debugging these applications. Since the space for representing event partial orders depends on the number of processes and events, it can grow quickly and become impractical to manage. The resource demands for such a

representation alone justifies the need for a more efficient representation of event partial orders.

2.4.2 Monitoring Requirements and Scalability

Monitoring and debugging large distributed applications present some unique constraints. Unlike application log data that can be read sequentially and easily partitioned for various offline data-mining tasks, monitoring and debugging facilities generally have stringent online requirements, are inherently centralized, and have complex event-data access patterns. We next discuss each of these requirements and the scalability problems that arise when working with large event-partial-order representations.

1. *Centralized and Online:* Monitoring and debugging are inherently centralized, as they need to take into account not just individual components, but also how these components interact with each other. This involves collecting information about the system and then performing various monitoring and debugging operations. Furthermore, Fidge/Mattern is a centralized algorithm for representing event partial orders, as it relies on the local clocks of every process for assigning timestamps to events. Another key feature of monitoring and control is that they are online and may run continuously for long periods of time. Therefore, the partial-order representation must be generated online and the representation-generation proces should be able to keep up with the target application under observation. Furthermore, if the size of the representation generated is large, the monitoring client would inevitably suffer when trying to query the partial-order representation.

2. *Partial-Order Access Patterns:* Various debugging and monitoring tasks including visualization, performance analysis, pattern search, distributed breakpoints, and event

abstraction typically perform the following queries on a partial-order representation of event data [69]:

- Looking up event information such as trace, type, text, and real-time
- Determining precedence between events
- Finding the greatest predecessors or least successors of events
- Looking up parter-event information
- Finding longest or shortest event paths

Many of these queries are performed on individual events or sets of relevant events that need to be accessed directly. Furthermore, as explored by Ward [69], the access patterns of many of these monitoring and debugging tasks generally result in poor temporal and spatial locality. This not only makes caching data ineffective, but can further result in thrashing in a virtual-memory system as the monitoring client becomes increasingly I/O-bound.

2.5 Summary

In summary, partial-order representation is essential for representing the relationships between events in a distributed application. This representation can, however, become very large as the number of processes increases. A direct consequence of this limitation is that it is extremely challenging to monitor and debug large distributed systems in real time. Therefore, efficiently representing event partial orders is not only critical from a resource-utilization standpoint but even moreso for facilitating online monitoring and debugging.

Chapter 3

Related Work

3.1 Introduction

Representing event relationships using partial orders is essential for monitoring and debugging distributed applications, however, as discussed in Chapter 2, naive representations of event partial orders do not scale. Furthermore, monitoring and debugging facilities have specific querying and event-data-access requirements. Based on these requirements, we can specify the following key features of a partial-order representation for events:

1. **Representation-Generation:** The dynamic or static nature and the computational complexity are the two critical aspects of a representation-generation scheme for event partial orders. We elaborate on each of these aspects below:

    - *Dynamic vs Static:* A dynamic representation of event partial orders can incorporate newly occurring events into the partial order as they are received by the

monitoring entity in an incremental fashion. On the other hand, a static algorithm requires access to all events before the partial-order representation can be constructed. As discussed in Chapter 2, monitoring is inherently online, and therefore, any scheme used for constructing the partial-order must be dynamic.

- *Computational Complexity:* The upper bound on the time required to generate the partial-order representation is also an important factor. A scheme for constructing the event partial order that is computationally expensive would quickly end up lagging behind the actual system under observation.

2. **Determining Precedence:** Testing precedence between two events is a basic operation and is carried out for many events for various tasks such as visualization, pattern search, and others. The computational complexity of precedence testing, therefore, is a critical aspect of any technique used for representing event partial orders.

3. **Space Efficiency and Event Access:** A key feature of any scheme for representing event partial orders is the space complexity. A closely related requirement is the cost of accessing partial-order information needed to determine precedence.

The features presented above offer a good starting point for comparing various existing techniques for representing event partial orders. We discuss these techniques in the next section.

3.2 Techniques

3.2.1 Trace-File Compression

A simple approach for reducing the space requirements of the partial-order representation is to compress the representation using a standard lossless data-compression technique such as gzip. Frumkin at el [27] improved on the compression that can be achieved by studying the information content of program traces. The information content is measured as the sum of the information-entropy [56] of the trace events, program communication, and timestamps. The authors show a storage efficiency of as high as 5 times that of original representation, however, it is not clear how the compressed representation can be used for precedence-testing. Furthermore, the compression technique cannot be used in an online setting.

3.2.2 Vector Clocks for Dynamic Systems

There are a number of techniques that rely on the dynamic nature of systems, i.e., the creation and termination of processes and threads, to conserve space when representing event partial orders. We describe two such techniques below:

Accordion Clocks

Accordion clocks [19] is a clock system specifically designed for detecting race conditions in parallel applications. The accordion clocks increases and shrink as threads are created and terminated in a parallel application. A data-race condition is defined as two events manipulating the same data in parallel, i.e., $e \parallel f$. As described in Chapter 2, determining

if two events on traces P_i and P_j are concurrent requires the comparison of only the ith and jth components of the Fidge/Mattern vector timestamp. The accordion-clock approach, therefore, throws away the components of a vector timestamp that correspond to the threads which no longer have any events of interest when detecting race conditions.

Interval Tree Clocks

Interval Tree Clocks [8] is a logical-clock system for highly dynamic systems. The clock system consists of three basic operations, namely, *fork*, *event*, and *join*. Fork clones an existing timestamp, creating a new copy of that timestamp. The new copy of the timestamp is assigned to a newly created trace that is forked from the original trace. The event operation increments a specific component of the timestamp as in Fidge/Mattern timestamps and the join operation merges two timestamps. A send event can be represented using an event operation, whereas a receive event is a join followed by an event operation. Similarly, a synchronous message is equivalent to a join followed by a fork. Interval Tree Clocks allow for completely decentralized creation of processes without the need for global identifiers. The mechanism has a variable-size representation that adapts automatically to the number of existing entities. The size of the timestamps grow with the number of forks for new processes and shrinks with the merge operations performed when processes terminate.

The approaches described above can be useful for specific tasks, such as data-race detection in parallel programs and version vectors for dynamic replica-generation; however, many distributed applications do not exhibit the level of dynamicity assumed in these techniques. In fact, distributed applications where a large number of processes are running simultaneously for significant time periods are very common. Furthermore, it is not clear

how precedence can be tested with timestamps where trace identifiers for old traces are reused for new traces.

3.2.3 Differential-Encoding-Based Techniques

When using Fidge/Mattern timestamps only a few components of the vector-timestamp change for successive events. This was exploited by Singhal and Kshemkalyani [52] for reducing the communication overhead when generating Fidge/Mattern timestamps in a distributed environment. Instead of sending the complete N-element vector timestamp with each message, a trace P_i sends to P_j only those components of the vector-timestamp that have changed since the last time P_i sent a message to P_j. The technique assumed FIFO communication channels. The original technique was improved by Hélary et al [**?**] to work without FIFO communication channels. Wang et al [65] further improved the differential encoding technique by taking into account processes starting and exiting in a dynamic system. Although these techniques can work well for generating vector timestamps in a distributed fashion by reducing the communication overhead, they don't directly address the problem of reducing the size of these timestamps.

In our work, we adapt the efficient partial-order representation schemes proposed by Taylor [61] to an online setting. The work proposes a number of novel differential encoding schemes for reducing the amount of data stored with each event when representing event partial orders. A significant advantage of the proposed scheme is that it can be readily adapted to an online setting without sacrificing space efficiency when representing event partial orders. We detail the scheme in Chapter 4.

3.2.4 Graph-Theoretic Approaches

A rich literature exists on graph-theoretic techniques that focus on maintaining dynamic transitive closures and efficient algorithms for dynamic reachability [**?**, 40, 54]. Recently, with the emergence of real-world applications, such as social-network analysis, semantic web (XML/RDF), and bio informatics, efficiently querying graphs has become an important research topic [38, 39, 74]. In graph databases, reachability is a fundamental query, i.e., given two vertices v_1 and v_2, does a path exist between them? For a partial-order representation of event orderings, precedence testing is equivalent to determining reachability between two vertices.

The research community has traditionally focused on the following key aspects of a representation scheme for graph databases:

1. **Query Time:** The computational complexity of a single reachability query. For our purposes this is the computational complexity of determining precedence between two events in a DAG representation of the event partial order.

2. **Index-Construction Time:** The time taken to create an index for the graph to quickly answer reachability queries. Again, this is equivalent to constructing a suitable partial order representation for answering precedence queries.

3. **Index Size:** The space required for the index or equivalently the space complexity of a graph-based partial-order representation.

Many of the existing techniques use simpler graph structures, such as chains and trees, to compress the transitive-closure for efficiently answering reachability queries. The approaches based on chain-decomposition and tree-cover are outlined as follows [38]:

The Chain-Decomposition Approach: In a chain-decomposition approach, a DAG is partitioned into pair-wise disjoint chains, i.e., each vertex in the graph can only be in a single chain. Each vertex is identified by a chain number c and a sequence number e. Note the uncanny similarities with the trace-based representation of events in a distributed application. The traces in a distributed application are naturally occurring chains and each event is uniquely identified by a trace identifier and an event sequence number. In chain-decomposition-based approaches, for each vertex v, one vertex u is recorded for each of the chains such that u is the smallest such vertex (sequence wise) reachable from v on that chain. In essence, chain-decomposition-based approaches maintain the *least successor* of an event on each of the traces. The least successor is defined as follows:

Definition 8 (Least Successor) *The least successor of an event, a, on trace P_i denoted $LS_{P_i}(a)$ is the single-element set containing the most-recent event, $\{e\}$, on trace P_i that happens after a, i.e., $a \rightarrow e$, or the empty set, $\{\}$, if no such event exists.*

Tree-Cover Approach: The tree-cover approach is based on interval labeling. Given a tree, a vertex v is assigned an interval $[i, j]$, where j is the postorder number of vertex v and i is the smallest postorder number among its descendants. If a vertex u can reach vertex v, then the interval of u contains the interval of v, therefore, checking if u can reach v, we only need to check if the interval of v is contained by the interval recorded for u.

Many of the existing approaches propose various changes to the above structures for improving query time, index construction, and index-space requirements. Earlier approaches focused on $O(1)$ query-time complexity at the expense of higher indexing-time and space complexities [**?**, 40]. A significant limitation, therefore, of these approaches is that they do not scale to large real-world graphs. This realization has led to a shift in focus towards more scalable indexing schemes. Two such recent schemes are GRAIL [74] and

	Query Time	Construction Time	Index Size
Transitive Closure	$O(1)$	$O(nm)$	$O(n^2)$
GRAIL	$O(d)$ to $O(n+m)$	$O(d(n+m))$	$O(dn)$
Path-tree	$O(log^2 k)$	$O(mk)$	$O(kn)$

Table 3.1: Comparison of GRAIL and Path-Tree

Path-tree [38]. The computational and space complexities of these schemes are shown in Table 3.1 for a graph with n vertices, m edges, k chains and d intervals (as outlined in the GRAIL paper [74]).

We next consider these schemes for representing event partial orders where we have N traces ($k = N$) and E events and the original graph is the transitive reduction of the partial order. As described in Chapter 2, the number of edges in the transitive reduction of the partial order is $m = E + M$, where M is the number of messages exchanged. We do not have the notion of intervals (d) in the Fidge/Mattern representation, however, in interval-based techniques, $d < k$ ($= N$). The updated complexities for the various schemes for representing the event partial order with N traces and E events are shown in Table 3.2.

GRAIL offers the most efficient representation of the partial-order with space complexity $O(dE)$ where $d < N$, better than the Fidge/Mattern approach, however, the query time is too high for our use case of monitoring and debugging large distributed applications. Path-tree on the other hand has worse construction-time complexity than the Fidge/-Mattern technique. The graph-theoretic approaches generally have higher construction costs because indexing graphs is a more generic problem than representing event partial orders. These schemes deal with graphs in general and not just DAGs. Additionally, graph-indexing schemes must support quick inserts and updates anywhere in the graph, whereas the requirements for representing event partial orders are less stringent. In an event-partial-order, only maximal events can be inserted and only the minimal events can

	Query Time	Construction Time	Index Size
Transitive Closure	$O(1)$	$O(E^2 + EM))$	$O(E^2)$
GRAIL	$O(d)$ to $O(E + M)$	$O(dE + dM))$	$O(dE)$
Path-tree	$O(log^2 N)$	$O(NE + NM))$	$O(NE)$
Fidge/Mattern	$O(1)$	$O(NE)$	$O(NE)$

Table 3.2: Comparison with Fidge/Mattern timestamps

be deleted. By restricting the requirements and taking advantage of the structure and communication patterns of distributed applications, the event partial orders can be represented more efficiently as we show in the next sections and in Chapter 4.

3.2.5 Dimension-Bound Ore Timestamps

At a minimum, vector clocks of size equal to the dimension of the partial order are required for determining the precedence relation between any two events in the partial order and it has been shown that the dimension of the partial order is bounded by the number of traces in a distributed application [16]. A distributed application with N traces would therefore need to attach an N-element vector timestamp to each event. Ward [67] showed that in practice the width of the partial order is often equal to the number of traces, however, in most cases the dimension of the event partial order is significantly smaller than the width. This motivated the development of a dynamic variant of Ore timestamps [68] that are bounded by the dimension and not by the width of the partial-order. We next describe the necessary partial-order terminology and the Ore timestamps, before discussing the dynamic Ore algorithm.

Definition 9 (Subposet) *A subposet $(Y, R_X|_Y)$ is a subset of poset (X, R_X) with a relation $R_X|_Y$ which is the restriction of the partial-order R_X to the set Y.*

Definition 10 (Extension) *An extension, (X, S_X), of a partial order (X, R_X) is any partial order that satisfies*

$$\forall_{x_1,x_2 \in X}\ (x_1, x_2) \in R_x \Rightarrow (x_1, x_2) \in S_x.$$

If S_X is a total order, then the extension is a *linear extension* or *linearization* of the partial order. Additionally, if $(Y, R|_Y)$ is a subposet of (X, R_X) and (Y, T_Y) is an extension of $R|_Y$ then (Y, T_Y) is called a *subextension* of (X, R_X).

Definition 11 (Realizer) *Given a poset (X, R_X) and a set $L = \{(X, L_X^i) \mid 0 \leq i < K\}$ of K linear extensions of the partial order, L forms a realizer of R_X if and only if*

$$R_X = \bigcap_i L_X^i.$$

A realizer of the partial-order is a set of linear extensions whose intersection is the original partial order. The dimension of the partial order is then simply the cardinality of the smallest realizer. The Ore timestamps [51] are based on the realizer of a partial order with d linear extensions. Each event e in each linear extension l_i of the realizer is assigned an id $l_i(e)$ to indicate the position of e in l_i. The following relation must hold for the position assigned to any events e and f in the linear extension l_i:

$$e \rightarrow_{l_i} f \Leftrightarrow l_i(e) < l_i(f) \tag{3.1}$$

Event e precedes event f in extension l_i if and only if $l_i(e) < l_i(f)$. The Ore timestamp for an event e is then the vector of positions of event e in all d extensions in the realizer, given formally:

$$\forall_{i:1 \leq i \leq d}\ Ore(e)[i] = l_i(e) \tag{3.2}$$

An event e precedes event f if and only if e precedes f in all linear extensions, that is

$$e \rightarrow f \Leftrightarrow \forall_{i:1 \leq i \leq d}\, Ore(e)[i] < Ore(f)[i] \tag{3.3}$$

Since computing a realizer of a partial order is NP-hard [73], the dimension-bound technique relies on an alternative result for the dimension of the partial order that relates to the *critical pairs* of a partial order.

Definition 12 (Critical Pair) *(x, y) is a critical pair of the partial order (X, R) if and only if $(x, y) \notin R_X$ and $(X, R \cup \{(x, y)\})$ is a partial order.*

A critical pair of the partial order is any pair not in the partial order, whose addition to the partial-order relation would result in another partial-order relation. Note that x is *covered by* y if there is no element between x and y in the partial order, i.e., no z exists in the poset such that (x, z) and (z, y) belong to the partial order. A significant result for the dimension of the partial order is given as follows:

Theorem 3 (Dimension) *The dimension of a partial order is the cardinality of the smallest possible set of subextensions that reverses all of the critical pairs of the partial order.*

The algorithm for assigning a dynamic Ore timestamp to each arriving event e consists of three steps. First, an iterative algorithm is used to compute the critical pairs for e, i.e., CP_e. The cost of this step is $O(N)$ where N is the number of traces. The next step is to reverse all the critical pairs in CP_e and insert them into extensions. The extensions need not be linear and therefore the realizer formed is referred to as a *pseudo-realizer*. If all critical pairs cannot be inserted into the existing extensions, a new extension is created. The computational complexity of this step is $O(kC)$, where k is a small constant and C

is the number of critical pairs for the event e, i.e., $|CP_e|$. The last step of the algorithm is to construct the dynamic Ore timestamp for the event e, which is simply the d-element vector containing the positions of e in each of the d extensions in the realizer. Note that the position of events can change as critical pairs are inserted in the extension which may require altering the position of large number of events. To prevent this from happening, each event is assigned a real number instead of an integer to mark its position in the extension.

The cost of determining precedence is $O(d)$ and the computational complexity for time-stamping E events using the dimension-bound scheme is $O((kC + N + d)E)$ which can be acceptable if k and C are relatively small. The scheme, however, is difficult to adapt to an online setting where often large number of critical pairs need to be reversed for each event. The space requirement for the dimension-bound timestamps is $O(d)$. For applications where the dimension of the partial-order is an order of magnitude smaller than the width of the partial-order, the scheme can be useful for saving space required for representing event partial orders, but, in practice, distributed applications can have high dimension. For such applications, the dimension-bound approach is of limited use.

3.2.6 Hierarchical Cluster Timestamps

Ward and Taylor [70, 71] proposed self-organizing hierarchical cluster timestamps that substantially modify the Summers cluster timestamps [59]. The motivation behind the approach is that there is communication locality in distributed applications, i.e., a trace tends to communicate mostly with only a few other traces. The communication in a distributed application can therefore be viewed as the communication within a group of traces and the communication between the different groups. These groups are referred

to as clusters and the key idea is to use a small vector equal to the number of traces in a cluster (size of cluster) for the timestamps of most events in a cluster. Finally, events within a cluster can causally depend on events outside the cluster, only if a message is sent from a trace outside the cluster to a trace within the cluster. The receive events for such messages are referred to as the *cluster-receive* events, defined as follows:

Definition 13 (Cluster-Receive) *An event e is a cluster-receive if and only if it is a receive event with a partner event on a trace in a different cluster or a synchronous event whose synchronous send and synchronous receive occur in different clusters.*

In a hierarchical cluster-based approach there can be k levels of clusters [71]. The timestamps of events in a cluster are of size $|c_k(e)|$ where $c_k(e)$ is the level-k cluster containing e. The timestamp size of a level-k cluster-receive event is $|c_{k+1}|$, i.e., the size of the level-$(k+1)$ cluster. The definition of cluster-receive can therefore be generalized as follows:

Definition 14 (Level-k Cluster-Receive) *An event e is a level-k cluster-receive if and only if it is a receive event with a partner event on a trace in a different level-k cluster or a synchronous event where synchronous send and synchronous receive occur on traces that are in two different level-k clusters.*

Note that by the above definition, a level-k cluster receive is also a level-0 to level-$(k-1)$ cluster-receive. The computational complexity of timestamping a level-k non-cluster-receive event e is $O(|c_k(e)|)$. For level-k cluster-receive events where k is near the top of the cluster hierarchy, the cost of computing the timestamp can be as high as $O(N|c_k(e)|)$, where N is the total number of traces. The computational complexity of

the precedence test depends on the level of cluster that encompasses both events being compared. If a level-k cluster encompasses both events, then the computational cost is $O(|c_{k-2}(e)||c_{k-1}(l)|)$, where $|c_w(e)|$ is the number of traces in the level-(w) cluster that contains e. Note that the size of the timestamps, the computational cost of timestamping each event and the cost of precedence testing depend on the size and the number of clusters. The size and the number of clusters in turn depend on the clustering strategy used for clustering traces in a distributed application. Ward and Taylor [70] explored a number of static and dynamic trace-clustering approaches, however, no single dynamic clustering technique works well for all distributed and parallel environments. This limits the use of hierarchical cluster timestamps in an online setting such as for monitoring purposes.

3.2.7 Summary

The techniques that take into consideration the structure and communication patterns of distributed and parallel applications, such as the dimension-bound Ore timestamps and the cluster timestamps are able to reduce the space required for a partial-order representation, however, it is difficult to adapt these schemes to an online setting. Furthermore, the cost of testing precedence is varied and can be high depending on the dimension of the partial order or the placement of traces in various clusters.

3.3 Tools for Monitoring and Debugging

Monitoring and debugging involve a number of facilities, e.g., visualization, event inspection, execution replay, and pattern search. Tools that provide some of these capabilities include XPVM [29] and ParaGraph [33]. In our work, we are using the Partial Order Event

Tracer (POET) [42], which is an existing tool built using many techniques and algorithms developed over the years.

POET itself is a distributed system with a client-server architecture. Figure 3.1 shows the architecture of the C++ variant of POET. The events from an application under observation (*target program*) are streamed to an event server. A number of different clients can then access these events to provide various monitoring and debugging capabilities. For example, a graphical-viewer client presents the partial-order relation between events to a user. Each trace is presented as a horizontal line and the relationships between events are presented using vertical or diagonal lines. Figure 3.2 shows the visualization for a sample distributed application. Since for most applications, all events cannot be displayed in a single window, a partial-order scrolling algorithm [60] was devised to present the correct partial-order view of traces as they are scrolled.

An advantage of using POET is that the client-server architecture allows for the development of various clients for exploring new algorithms and techniques, such as online trace-reordering algorithms for efficiently representing event partial orders. Another significant advantage of using POET is that it is target-system independent and therefore can be used to monitor and debug applications in many different environments. This capability allows us to explore the effectiveness of online trace-reordering schemes on many different target applications. The original version of POET was written in C and stored events in a complex flat file, however, we are working with a more recent C++ variant of the tool that stores events in a relational database. The efficient implementation scheme proposed by Taylor [61] is built as a separate client in the C POET. We have ported this existing functionality into the C++ variant of POET and extended it by developing a number of online trace-reordering schemes.

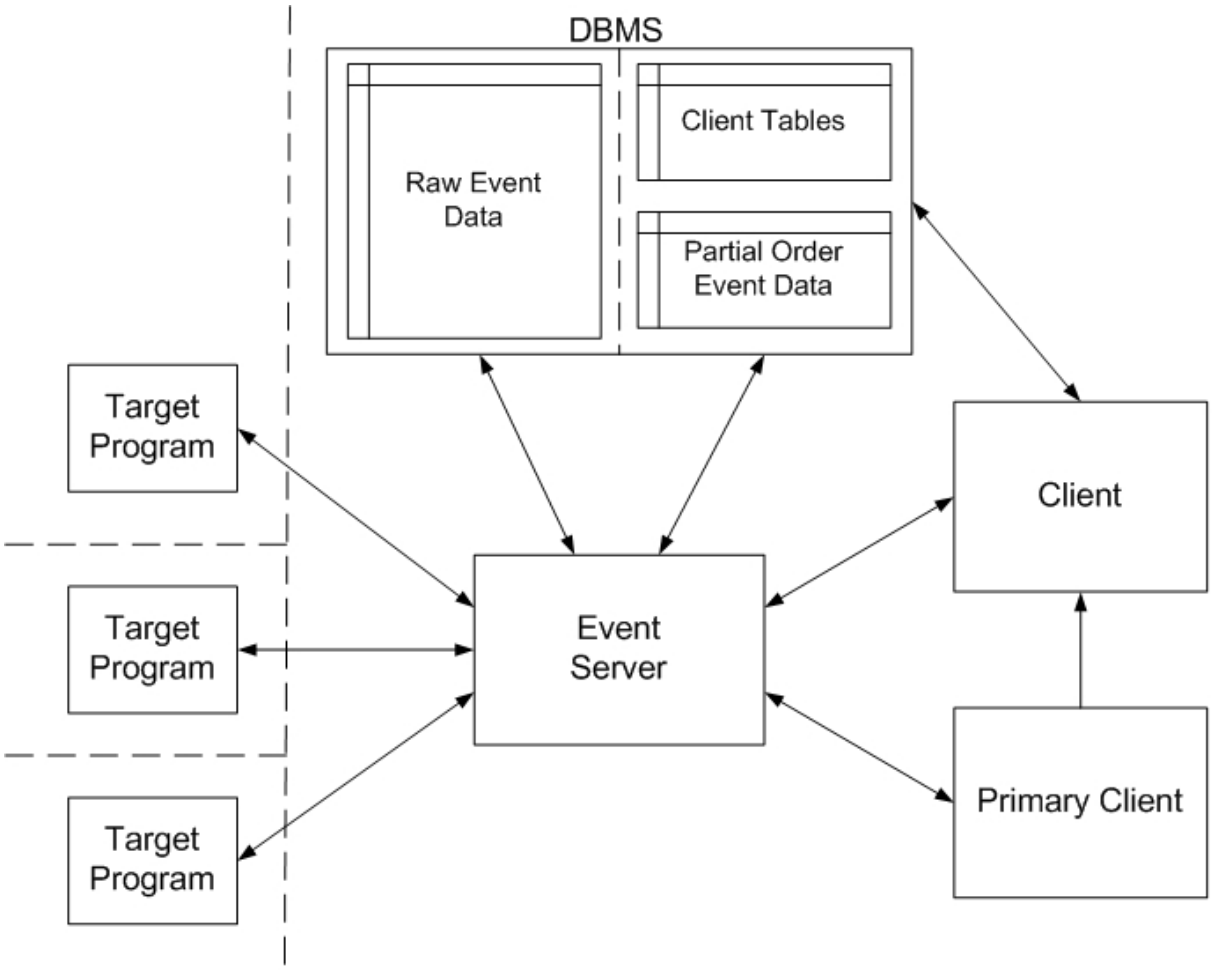

Figure 3.1: C++ POET architecture

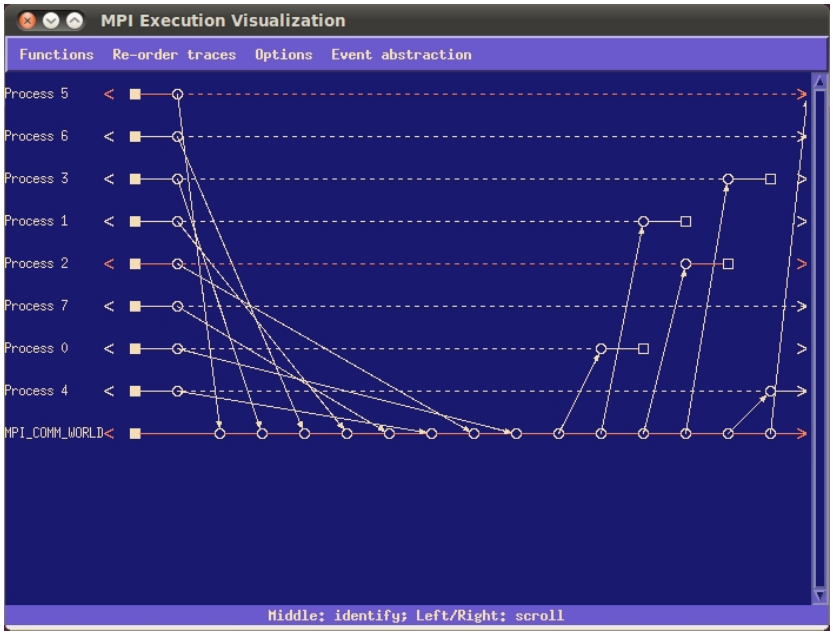

Figure 3.2: POET GUI-Viewer client

Chapter 4

Efficient Representation of Event Partial Orders

4.1 Introduction

In our work we develop a number of online trace-reordering schemes for the offset-based representation of event partial orders developed by Taylor [61]. Timestamps of successive events on a trace are closely related, for example, the vector timestamp of a unary or an asynchronous send event differs from the timestamp of the last event on that trace in only a single vector element. A synchronous event or an asynchronous receive event on the other hand can have a timestamp that differs significantly from the timestamp of the last event, i.e., many of the vector-timestamp elements of the two events differ. Such significant changes do occur, but are quite infrequent due to the communication locality exhibited by distributed and parallel applications. The offset-based representation scheme captures this communication locality (when present) to efficiently represent event partial orders without

explicitly depending on the communication pattern exhibited by the application. This is in contrast to the earlier approaches, such as the Hierarchical Cluster Timestamps [70] where for example, no single clustering scheme works for all applications because of the variations in communication patterns of such applications.

In offset-based representation of event partial orders a number of Fidge/Mattern timestamps are maintained in a global cache. Each event maintains a number of fixed-sized *offsets* and a reference to one of the timestamps in the cache. These offsets can then be used to transform the referenced timestamp in cache into the event's Fidge/Mattern timestamp. The Fidge/Mattern timestamps maintained in the global cache are referred to as the *base timestamps* and the global cache is referred to as the *base-timestamp cache* or simply as the *cache*. In the next section we describe three different schemes used for computing the offsets for an event relative to a base timestamp [61].

4.2 Offset-Based Representation Schemes

4.2.1 Individual Differences

In this scheme, each event e stores the individual differences of e's timestamp (T_e) from one of the base timestamps T_b in cache, i.e., a number of (i, v) offsets are stored for event e such that

$$T_e[i] - T_b[i] = v \tag{4.1}$$

Consider for example a base timestamp T_b and a Fidge/Mattern timestamp T_e of an event for a 20-trace application:

$i = 0$ 4 10 13 19

T_b: 0, 0, 0, 1, 2, 2, 0, 1, 4, 5, 3, 0, 0, 1, 0, 0, 0, 0, 1, 3

T_e: 1, 0, 0, 1, 0, 2, 0, 1, 4, 5, 8, 0, 0, 0, 0, 0, 0, 0, 1, 4

The timestamp T_e of event e can thus be completely constructed from T_b by maintaining the following vector of individual differences:

$$< (0, 1), (4, -2), (10, 5), (19, 1) >$$

For the individual-differences scheme, the size of each offset is $S_{off} = 2 \times sizeof(int) = 8$ bytes (we assume a 4-byte integer throughout). We can therefore save space by storing a reference R_{T_b} to the base timestamp T_b and the offsets $off_{T_b}(e)$ for event e relative to T_b. For the example above, we would require $R_{T_b} + |off_{T_b}(e)| \times S_{off} = 4 + 4 \times 8 = 36$ bytes of space instead of the 80 bytes required for storing the 20-element Fidge/Mattern timestamp T_e for event e.

4.2.2 Identical Differences

The identical-differences scheme records a series of individual differences together if they are identical. A vector of triples $< (i, j, v) >$ is maintained such that the timestamp T_e of an event e differs from a base timestamp T_b by v for traces i through j, i.e.,

$$\forall_{i \le k \le j}\ T_e[k] - T_b[k] = v \tag{4.2}$$

Consider the following base timestamp T_b and an event e with timestamp T_e:

$i = 0$ — 2 4 8 ——— 12 16 — 19

T_b: 0, 0, 0, 1, 2, 2, 0, 1, 4, 5, 3, 0, 0, 1, 0, 0, 0, 0, 1, 3

T_e: 2, 2, 2, 1, 0, 2, 0, 1, 5, 6, 4, 1, 1, 1, 0, 0, 3, 3, 4, 6

The offsets using the identical differences scheme are

$$\mathit{off}_{T_b}(e) : < (0, 2, 2),\ (4, 4, -2),\ (8, 12, 1),\ (16, 19, 3) >$$

Note that the size of each offset (S_{off}) for the identical-differences scheme is 12 bytes. In the above example, the space required using the identical-differences scheme is $4 + 12 \times 4 = 52$ bytes. The reader can verify that the space required using the individual-differences scheme is 108 bytes (more than the 80 bytes for the complete Fidge/Mattern timestamp T_e).

4.2.3 Incremented Differences

The incremented-differences scheme records a sequence of individual differences such that the sequence follows an arithmetic progression. A vector of four-tuples $< (i, j, v, q) >$ is maintained for an event e where T_e differs from a base timestamp T_b by a sequence of differences from trace i to trace j, i.e.,

$$\forall_{i \le k \le j}\ T_e[k] - T_b[k] = v + (k - i)q \tag{4.3}$$

$i =$	1	—	3	4			7	——	10								16	–	18
T_b:	0,	0,	0,	1,	2,	2,	0,	1,	4,	5,	3,	0,	0,	1,	0,	0,	0,	0,	1,	3
T_e:	0,	3,	2,	2,	0,	2,	0,	0,	4,	6,	5,	0,	0,	1,	0,	0,	3,	3,	4,	3

For the base timestamp T_b and the event e with timestamp T_e as shown above, the vector of offsets for the incremented-differences scheme is

$$\mathit{off}_{T_b}(e) : < (1, 3, 3, -1),\ (4, 4, 2, 0),\ (7, 10, -1, 1),\ (16, 18, 3, 0) >$$

The size of each offset S_{off} is 16 bytes and the space required for representing event e using the incremented-differences scheme is 68 bytes. Alternatively, the space required using the individual-differences scheme and the identical-differences scheme is 84 bytes and 100 bytes respectively.

4.3 Generating Offset-Based Representation

When a new event e arrives, several are taken (following Algorithm 1) to generate the offset-based representation for e. First, the Fidge/Mattern timestamp T_e is computed for event e. A base timestamp T_b is picked from the base-timestamp cache and the offsets of T_e are computed relative to T_b using one of the schemes described above. If the number of offsets $|off_{T_b}(e)|$ is within a pre-defined *OFFSET_LIMIT* (line 5), the offsets $off_{T_b}(e)$ and the reference to the base timestamp R_{T_b} are saved for the event e. If the number of offsets is more than the OFFSET_LIMIT, the next base timestamp from cache is picked and the process is repeated until a base timestamp is found that can be used to successfully represent event e (loop from line 3 to 8). If all base timestamps in the cache are exhausted without success, the timestamp T_e of event e is saved as a new base timestamp (line 10). T_e is also added to the base-timestamp cache (line 11) and a reference to T_e is stored for event e with no offsets (line 12). If the base timestamp cache is full, the least recently used base timestamp is removed from the cache to make room for the new base timestamp. Note that the OFFSET_LIMIT, the offset scheme to use, and the size of the base-timestamp cache are specified as configuration parameters.

Algorithm 1 OffsetRepresentation(Event e, Scheme $scheme$)

1: $T_e \leftarrow FidgeMattern(e)$
2: $bId \leftarrow -1$, $cacheIndex \leftarrow 0$, $offsets \leftarrow [\,]$
3: **for** $cacheIndex < length(cache_B)$ **do**
4: $\quad offsets \leftarrow GenerateOffsets(T_e, cache_B[cacheIndex], scheme)$
5: $\quad$ **if** $length(offsets) \leq OFFSET\_LIMIT$ **then**
6: $\quad\quad$ **break**
7: $\quad$ **end if**
8: **end for**
9: **if** $length(cache_B) = 0$ **or** $cacheIndex \geq length(cache_B)$ **then**
10: $\quad bId \leftarrow StoreBaseTimeStamp(T_e)$
11: $\quad InsertInCache(T_e)$
12: $\quad StoreEventRepresentation(e, offsets, bId)$
13: **else**
14: $\quad bId = GetBaseTimestampId(cache_B[cacheIndex])$
15: $\quad StoreEventRepresentation(e, offsets, bId)$
16: **end if**

4.3.1 Computational Complexity

The computational complexity of generating a Fidge/Mattern timestamp is $O(N)$ where N is the number of traces. Generating offsets for an event relative to a base timestamp requires a traversal of the Fidge/Mattern timestamp and therefore is $O(N)$ (line 4). The overall complexity for generating the offset-base representation depends on the number of base timestamps that were checked for an event, i.e., the iterations of the *for loop* in Algorithm 1. The time complexity thus depends on the number of base timestamps that are tried for an event e, i.e., $B_{search}(e)$. In the worst case scenario we may end up trying every base timestamp in a full cache and then create a new base timestamp. In such a case $B_{search} = CACHE\_SIZE$ where CACHE_SIZE is the maximum size of cache and is a configurable parameter. The complexity for generating the offset-based representation for E events is therefore $O(AVG(B_{search}) \times NE)$ where $AVG(B_{search})$ is the average number

of base timestamps searched for the E events, i.e., $\frac{1}{E}\sum_{e} B_{search}(e)$.

4.3.2 Space Complexity

The space (in bytes) required for the offset-based representation is equal to the space required for all the base timestamps and the space required for all the offsets and the references (to base timestamps) maintained for all events, i.e.,

$$Representation\ Bytes = 4 \times BN + E \times (R_b + AVG(Offs) \times S_{off}) \tag{4.4}$$

where B is the total number of base timestamps, E is the total number of events, R_b is the size of a single reference to a base timestamp (always 4 bytes), $AVG(Offs)$ is the average number of offsets, i.e., $AVG(Offs) = \frac{1}{E}\sum_{e} off_{T_b}(e)$, and S_{off} is the size of each offset which can be 8, 12 or 16 bytes depending on the offset scheme used. Thus the worst case space complexity is $O(BN + OFFSET\_LIMIT \times (E - B))$, where OFFSET_LIMIT is the maximum number of offsets that can be used for representing a single event.

For the offset-based representation to be useful in saving space, the space required for each event that is successfully represented using just the offsets must be less than the space required for storing the Fidge/Mattern timestamp for that event, thus the following inequality gives an approximate upper bound on OFFSET_LIMIT for an N-trace application where $E >> N$:

$$OFFSET\_LIMIT \times S_{off} < 4 \times N \tag{4.5}$$

For example, using the incremented-sequence scheme (16 byte offset) for a 100-trace application, in no case should the maximum number of offsets used for an event be more than 24. Taylor [61], however, showed that in practice the number of offsets that is required, even for large applications, is significantly less than this upper bound.

The total bytes stored for all events is the sum of a fixed number of bytes for an event and the bytes used for the partial-order representation. In POET [**?**], a fixed 28 bytes of space is used for an event. Thus the total space required (in bytes) is given by

$$\begin{aligned} \mathit{Total\ Event\ Bytes} &= \mathit{Fixed\ Event\ Bytes} + \mathit{Representation\ Bytes} \\ &= (28 \times E) + (4 \times BN + E \times (4 + AVG(\mathit{Offs}) \times S_{\mathit{off}})) \end{aligned} \tag{4.6}$$

Whereas the total bytes using the Fidge/Mattern scheme is given by

$$\begin{aligned} \mathit{Total\ Event\ Bytes} &= \mathit{Fixed\ Event\ Bytes} + \mathit{Fidge/Mattern\ Bytes} \\ &= (28 \times E) + (4 \times EN) \end{aligned} \tag{4.7}$$

4.3.3 Precedence Testing

The precedence between two events e and f on traces i and j can simply be tested by comparing the respective Fidge/Mattern timestamp components, i.e., to check if $e \rightarrow f$ we need to compare $T_e[i]$ and $T_f[i]$. Since each event is identified uniquely by its trace and the sequence number on that trace, for event e on trace i, $T_e[i]$ would simply be the sequence number stored as part of the event identifier, therefore, we only need to compute $T_f[i]$ from its offset-based representation. The precedence test is thus given by Algorithm 2. The computational complexity of determining precedence simply depends on

Algorithm 2 PrecedenceTest(Event e, Event f, Scheme *scheme*)

```
1: fSeqNum ← GetFidgeMatternVectorComponent(f, e.trace, scheme)
2: if e.seqNum < fSeqNum then
3:    return true
4: else
5:    return false
6: end if
```

the cost of computing $T_f[i]$ (line 1). $T_f[i]$ can be computed by checking each offset of f

until an offset is found that affects trace i of T_f. In the case where no offset affects trace i, $T_f[i] = T_b[i]$ where T_b is the base timestamp referenced by f, therefore, the worst-case complexity of the algorithm is $O(OFFSET\_LIMIT)$, i.e., when all the offsets of an event are checked. For most real applications an OFFSET_LIMIT of 4 is sufficient [61], therefore, the precedence test is very efficient with a computational complexity that is constant for practical purposes.

4.4 Analysis of Schemes

The comparison of the three offset-based schemes [61] showed that the incremented-sequence scheme was the most effective in reducing the space required for representing the event partial order for a number of distributed and parallel applications. We therefore only use the incremented-sequence scheme for online trace-reordering for the offset-based representation.

4.4.1 Order of Traces

Although the offset-based schemes are effective in reducing the space required for representing event partial orders, Taylor [61] showed that the space efficiency and the computational cost can improve significantly if traces that communicate with each other are close to each other. In some applications the natural order in which traces are created is also the best order based on the communication between traces, however, for many applications this is not the case.

POET already has a number of heuristic-based algorithms for ordering traces that were developed for better visualization. The two key aspects of the algorithms are the order in

which traces are "processed" and the cost function of a trace relative to another trace. For example, for an N-trace application with traces $T_1, T_2, \ldots T_N$, the goal is to put all N traces in a new order e.g. $T_7, T_{21}, T_1, \ldots T_{16}$, such that the traces that communicate with each other are close to each other. Each trace is picked and put into its final place in the new trace order based on a cost function and the level of communication with other traces. The order in which a trace is picked by the algorithm can be based on the total communication of all traces with that trace or alternatively on the number of already processed traces that are directly connected to this trace (by a message). Once a trace is selected, the cost function is used to determine the position of the trace in the new order. For example, consider two traces T_i and T_j where they exchange m_{ij} messages. The cost of T_i and T_j relative to each other is then

$$cost_{ij} = cost\ function \times m_{ij} \tag{4.8}$$

The goal of the trace-reordering algorithm is to minimize the total cost for all traces relative to all other traces:

$$total\ cost = \sum_{i=1}^{N} \sum_{j=i+1}^{N} cost_{ij} \tag{4.9}$$

Two possible cost functions are the trace-distance cost function and the 0-1 cost function, given as follows:

$$dist\_cost(i,j) = \begin{cases} |i-j|-1, & \text{if } i \neq j \\ 0, & \text{if } i = j \end{cases} \tag{4.10}$$

$$01\_cost(i,j) = \begin{cases} 0, & \text{if } |i-j| \leq 1 \\ 1, & \text{otherwise} \end{cases} \tag{4.11}$$

The trace-reordering algorithm that resulted in the most efficient representation was the variant which processed the traces based on the highest number of connections to already

processed traces and used a 0-1 cost function [61]. In the classic POET, the traces are reordered after seeing all the events and the trace order can then be used for the offset-based representation of the event partial order. Thus, although the offset-based representation scheme works online in classic POET, the additional space and computational efficiency that can be achieved by reordering traces is not available in an online setting. In our work, we directly extend the offset-based representation scheme to order traces dynamically and evaluate a number of such online trace-reordering schemes.

4.4.2 Parameter Selection

Before we can use the incremented-sequence scheme for representing event partial orders, there are two configuration parameters that must be specified, i.e., the base-timestamp-cache size CACHE_SIZE and the maximum number of offsets that can be used for representing an event (OFFSET_LIMIT). We discuss these parameters in order:

Cache Size

The cache size dictates the maximum number of base timestamps that are searched for a suitable base-timestamp match for representing an event. A very small cache may result in a lot of "cache misses" and therefore would result in the creation of large number of base timestamps B. Since each base timestamp is $4N$ bytes, too many base timestamps will result in higher number of bytes stored per event (Equation 4.4). On the other hand, a large cache will result in higher search times associated with a "cache miss", i.e., higher $AVG(B_{search})$, which translates to higher computational cost ($O((AVG(B_{search})+1) \times N)$) per event.

The analysis of distributed and parallel applications [61] shows that only the most recently used base timestamps are likely to be a successful fit for representing a new event. Furthermore, a cache size of approximately one-fourth the number of traces (in the application) is sufficient for almost all applications. The cache size can therefore be dynamically adjusted based on the number of traces in the application, i.e., setting it to $\frac{N}{4}$ where N is the number of traces. Since in our analysis of online trace-reordering schemes, the largest application we consider has around 1000 traces, we fix the value of CACHE_SIZE to 256. This is a small enough space requirement, taking as much as $256 \times N \times 4$ bytes, which is approximately 1MB of memory.

Offset Limit

The maximum number of offsets allowed for an event (OFFSET_LIMIT) is the configurable parameter of most significance for offset-based schemes. From a computational perspective, one might think that a large value of OFFSET_LIMIT would result in smaller search for a suitable base timestamp. Although this is the case when the number of allowed offsets is too few, i.e., less than 4, the results [61] show that the $AVG(B_{search})$ quickly drops to almost 1 when $OFFSET\_LIMIT \geq 4$ for most applications and therefore, a further increase in the number of offsets translates into diminishing returns for $AVG(B_{search})$. Note that the computational complexity of offset-based schemes ($O((AVG(B_{search}) + 1) \times N)$) with $AVG(B_{search}) = 1$ is twice that of the Fidge/Mattern scheme ($O(N)$).

We next discuss the impact of OFFSET_LIMIT on the space required for the representation. Note that in Equation 4.4, the bytes required for the partial-order representation not only depends on the number of base timestamps B, but also on the average number of offsets $AVG(\mathit{Offs})$ stored for each event. The results [61] show that 3 offsets generally result in the smallest number of representation bytes per event. For fewer than 3 offsets

per event, the number of base timestamps B is too high resulting in higher bytes/event. Similarly, for an OFFSET_LIMIT of greater than 3, the bytes required for event offsets overshadows the bytes required for the base timestamps, resulting in higher bytes/event. However, as discussed above, 4 offsets generally result in a very efficient representation-generation process, therefore, we sacrifice little space efficiency for considerable gains in computational cost and use an OFFSET_LIMIT of 4 for online trace-reordering schemes.

4.5 Summary

The offset-based representation schemes presented in [61] make minor trade-offs in computational cost relative to the Fidge/Mattern scheme and achieve significant space saving, often on the order of 2 to 7 compared to the Fidge/Mattern approach and can be readily used in an online setting. In Chapter 5, we propose and evaluate a number of online trace-reordering schemes and compare the space-efficiency and the computational cost of the offset-based representation with online trace reordering compared with the representation scheme with offline trace reordering.

Chapter 5

Online Trace Reordering

As mentioned in Chapter 4, we ported the offset-based representation from classic POET to C++ POET. We first detail the implementation and then describe a number of schemes used for online trace reordering.

5.1 Implementation

In POET a client can be designated as a *primary client* which is then eligible to be the first client started by the server. The server automatically picks the first client from the list of primary clients and runs it each time the server starts. In classic POET the viewer client is first primary client and is run when the server starts; a console client is an alternative primary client. In C++ POET, a console client is the first primary client and can be used for running the desired target programs and clients. We have developed a layered architecture for the development of clients that allows various clients to re-use functionality of existing clients. We describe the client architecture in the next section.

5.1.1 Layered Client Architecture

C++ POET has a number of client APIs which are part of the *client-stub* library. The client stub library provides functionality that is required by all clients, such as the core API for communicating with the server. The client stub also maintains a small cache of events and event counters for each trace. Additionally, the client stub maintains the *event table* that is used for keeping track of event types and provides an API for checking event types. The CLIENT class maintains basic information such as user, host, and client stub and provides an interface that we expect most clients to follow with the exception of few special cases. The basic CLIENT interface can be implemented by various clients according to their own requirements and is given as follows:

```
1  class CLIENT{
       public:
       inline virtual int pre_connect(){ return 0; };
4      virtual int connect() = 0;
       inline virtual int post_connect(){ return 0; };
       inline virtual int register_callbacks(){ return 0; };
7      inline virtual int main_loop(){ return 0; };
       inline virtual int shutdown(){ return 0; };
       inline virtual int initialize(){
10             pre_connect();
               connect();
               post_connect();
13             register_callbacks();
               return 0;
       };
16     inline virtual int run(){
               initialize();
               main_loop();
19             shutdown();
               return 0;
```

```
            };
22          ...
            ...
}
```

Many clients follow a workflow where they connect to the server and poll the server periodically for new events until they receive the *shutdown* signal from the server. The POET server requires that all clients must first be registered with it using a *client registration service*. Each time a client establishes a connection to the server, it provides its registration identifier. Thus, in the CLIENT class, *connect* (line 6) is a pure virtual function that must be implemented by all clients. The client functionality is mainly divided into two phases, the initialization phase and the main execution and termination phase. The initialization phase cosists of a number of steps like setting up data structures before connecting to the server and then getting the necessary information from the server, such as the client table and the number of processes, after connecting to the server. This is normally followed by registering for events that are of interest to the client. These steps form the *pre_connect, connect, post_connect,* and *register_callback* functionality of the client. By segmenting the client functionality into a sequence of actions, any client can provide only the functionality that is unique to it and re-use much of the existing functionality of other clients. We modified the primary console client to implement the CLIENT interface described above where it implements its own *connect* and *main loop*. Instead of polling the server for events, the console client reads user-input using a *command-line manager* and performs different tasks in response, including starting a new client and running a target application for monitoring.

5.1.2 Offset-Based Representation Client

The first step of the offset-based representation scheme is to generate the Fidge/Mattern timestamp of an event, therefore, we ported the Fidge/Mattern-timestamp-generation capability of classic POET to C++ POET by implementing a TS_CLIENT. The TS_CLIENT maintains a TIMESTAMPER object that encapsulates the functionality of generating Fidge/Mattern timestamps. Thus, for the TS_CLIENT, the *main loop* is simply given as follows:

```
int TS_CLIENT::main_loop(){
        while(1) {
                client_stub.wait_refresh();
                if (client_stub.should_shutdown())
                        shut_down();
                timestamper.timestamp((TS_EVENT *) 0);
        }
        return 0;
}
```

The *timestamper* goes through each event, generating its Fidge/Mattern timestamp, until no more events are available. The offset-based representation functionality proposed by Taylor [61] is ported to C++ POET by implementing a TS_DIFF_CLIENT that extends the TS_CLIENT. Once an event is timestamped, i.e., its Fidge/Mattern timestamp is generated, the TS_DIFF_CLIENT gets that event along with its timestamp to generate the offset-based representation. Note that the TS_DIFF_CLIENT does not provide its own *main loop* since it is the same as for the TS_CLIENT. The class diagram representing the more significant components used by the TS_DIFF_CLIENT is shown in Figure 5.1.

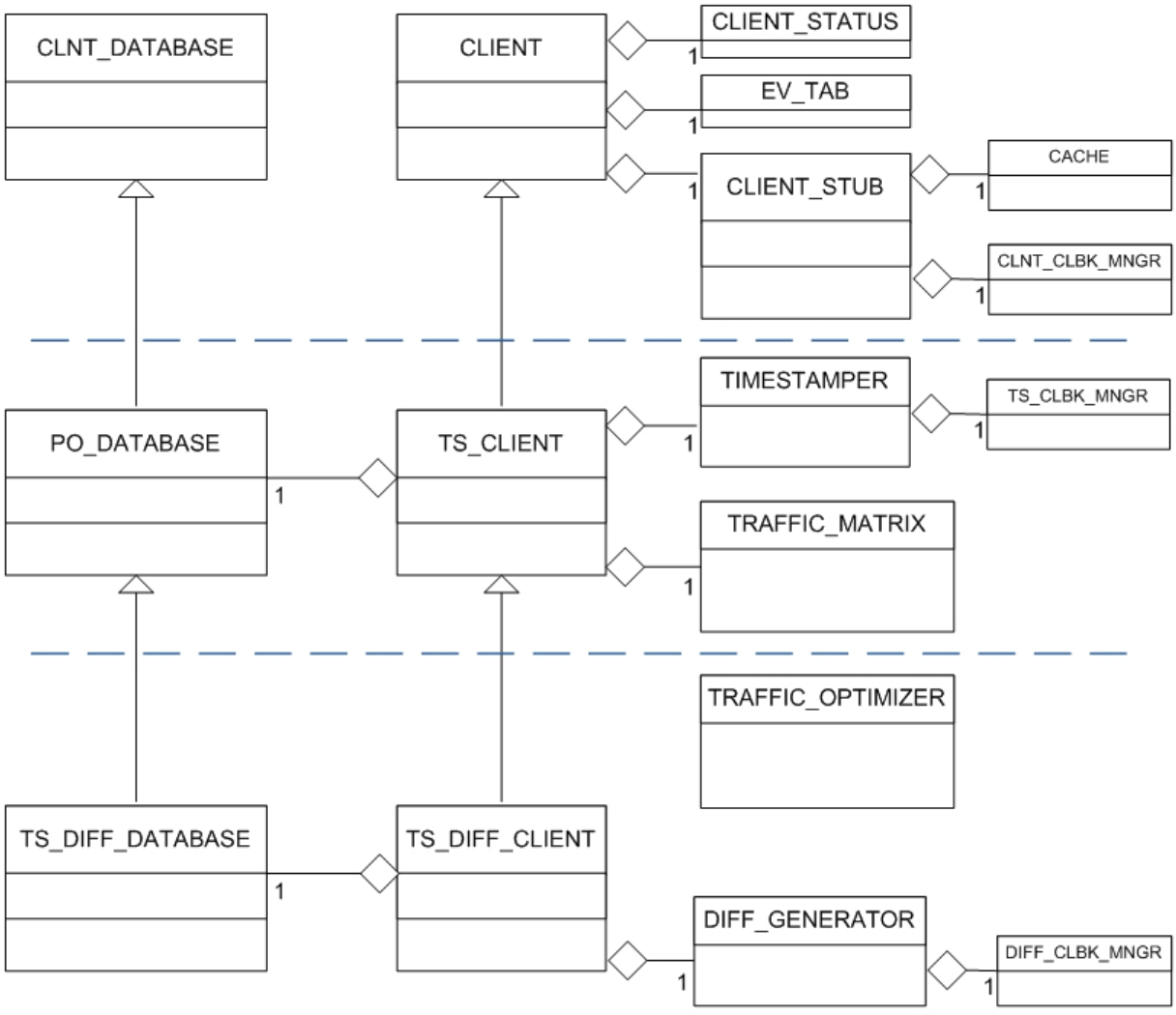

Figure 5.1: Class diagram for offset-based representation client

The Callback Mechanism

A callback mechanism is implemented to facilitate a client to perform client-specific tasks when certain actions are performed. The TS_CLBK_MNGR (shown in Figure 5.1) registers callbacks for actions such as when an event is timestamped by the *timestamper*. The TS_DIFF_CLIENT registers its callbacks with this callback manager to get an event's Fidge/Mattern timestamp. Any client that is built on top of the TS_CLIENT, i.e., inherits TS_CLIENT, can either add its own callbacks to the callbacks already registered by the TS_CLIENT layer or can register its own callbacks while discarding any callbacks registered by the TS_CLIENT layer simply by providing its own implementation of the *register_callbacks* method. In our case, it is natural to augment the callbacks already registered by the TS_CLIENT layer in the TS_DIFF_CLIENT. The register_callbacks for TS_DIFF_CLIENT is thus simply:

```
int TS_DIFF_CLIENT::register_callbacks(){
        TS_CLIENT::register_callbacks();
3       ts_clbk_mngr->register_do_event_clbk(fn_do_event);
        ts_clbk_mngr->register_wait_refresh_clbk(fn_wait_refresh);
        return 0;
6 }
```

Ordering Traces Based on Traffic Matrix

The TRAFFIC_MATRIX maintains an $N \times N$ matrix for keeping track of the number of interactions between any two traces i and j. Note that the traffic matrix is a symmetric matrix. The TS_CLIENT updates an instance of the traffic matrix when an event is timestamped by the timestamper. The TRAFFIC_MATRIX keeps track of the updates to the matrix and periodically saves the changes to a persistent data store. The

TRAFFIC_OPTIMIZER provides a number of *static* methods that take an instance of TRAFFIC_MATRIX and return the *trace order* based on one of the schemes described in Chapter 4. A trace order is an N-element vector where N is the number of traces and the position of a trace i in the new order of traces is recorded against its index in the vector.

Database Storage

Several classes are provided for accessing and storing event representations in the database. The CLNT_DATABASE provides an API for accessing basic information about events and traces. The partial-order database (PO_DATABASE) provides the API for storing and retrieving the Fidge/Mattern timestamps and the traffic matrix. Similarly, the TS_DIFF_DATABASE provides the additional API for storing and accessing offset-based representations of events. For security purposes, an instance of a client database that is inherited from CLNT_DATABASE does not have sufficient information to directly connect to the database server and therefore, relies on the client-stub API for establishing a connection to the database server.

5.2 No Trace Reordering

We start by analyzing the space required for an offset-based partial order representation if traces are not reordered, i.e., the order in which the traces are created in the target application is used as is for generating the representation. Table 5.1 shows the space required for representing event partial orders using Fidge/Mattern timestamps and Taylor's offset-based representation [61] for a number of MPI [34] applications.

Application	Traces	Events	Event Bytes	(Event + PO Bytes)/Event	
				Fidge/Mattern	Offset-Scheme
Life	250	502500	14070000	1028	100
ParallelLife	251	101502	2842056	1032	265
ParallelLife	501	203002	5684056	2032	759
ParallelLife	1001	406002	11368056	4032	1699
Nbody	251	1256502	35182056	1032	222
Nbody	501	5013002	140364056	2032	364
Nbody	1001	20026002	560728056	4032	729
WaveSend	251	500248	14006944	1032	353
WaveSend	501	1002498	28069944	2032	706
WaveSend	1001	2006998	56195944	4032	1570
WaveShift	251	500248	14006944	1032	587
WaveShift	501	1002498	28069944	2032	1222
WaveShift	1001	2006998	56195944	4032	2763
Random	251	53248	1490944	1032	519

Table 5.1: Bytes/event for Fidge/Mattern and Taylor's offset-based representation

Note that the CACHE_SIZE is set to 256 and the OFFSET_LIMIT for representing a single event is set to 4. When generating offset-based representations online, we saw small variations in the total space required for representation. The variation can be attributed to the dynamic nature of the application under observation and the order in which different events are received by the client and, hence, the order in which timestamps are generated. Overall, the variations we saw were quite small, less than 10% for almost all applications (presented in Section 5.4.1).

Although applications do see significant space savings even without trace reordering, Taylor [61] showed that further space saving can be achieved for the rest of the applications by reordering traces. Additionally, the overall computational complexity can be significantly reduced by getting the right order of traces. As described in Chapter 4, the computational complexity of generating the representation is given by $O((AVG(B_{search}) + 1)N)$,

Application	Traces	Events	Base Timestamps	Avg. Search
Life	250	502500	1882	62.6
ParallelLife	251	101502	17012	74.0
ParallelLife	501	203002	67158	99.1
ParallelLife	1001	406002	162549	109.9
Nbody	251	1256502	157622	42.1
Nbody	501	5013002	669815	43.8
Nbody	1001	20026002	3164858	53.7
WaveSend	251	500248	127296	80.8
WaveSend	501	1002498	305121	94.6
WaveSend	1001	2006998	738608	104.0
WaveShift	251	500248	244751	140.2
WaveShift	501	1002498	563351	153.8
WaveShift	1001	2006998	1336716	175.6
Random	251	53248	22494	119.4

Table 5.2: No trace reorder with CACHE_SIZE of 256 and OFFSET_LIMIT of 4

where $AVG(B_{search})$ is the average number of base timestamps that are searched for representing a single event. Table 5.2 shows that the average search is quite high, making the computational cost of the approach significantly higher than Fidge/Mattern approach. Furthermore, to efficiently determine precedence between two events and to recompute timestamps for events using their offsets, all base timestamps are cached in memory. Thus, reducing the total number of base timestamps can significantly reduce the in-memory overhead of using the offset-based scheme. These results motivate the need for online trace reordering for Taylor's offset-based representation scheme.

5.3 Online Trace Reordering

In this section we propose and evaluate a number of schemes for ordering traces online for Taylor's offset-based representation. We then compare the storage and computational cost

of the online representation schemes with the results presented in the original work [61], where a single offline trace order is used. Note that to keep the comparison consistent we report the space required for storing the representation in a flat-file format. In Section 5.4.3, we compare the space required for storing the representation in a relational database.

In the online setting, to accommodate multiple trace orderings (also referred to as trace permutations), an additional cache is maintained for trace orders. An event not only maintains a pointer to a specific base timestamp but also a pointer to a specific permutation. Therefore, the total space required for the online offset-based representation includes the space required for storing all the permutations and the 4 bytes for the additional pointer maintained by each event. The total space in bytes is:

$$Total\ Event\ Bytes = (28 \times E) + (4 \times (B + P)N + E \times (8 + AVG(\mathit{Offs}) \times S_{off})) \quad (5.1)$$

Where E, N, B and P are the total number of events, traces, base timestamps, and permutations respectively. $AVG(\mathit{Offs})$ is the average number of offsets for each event and S_{off} is the size of each offset (16 bytes for the incremented-sequence scheme). Note that for the flat-file-based storage, to ensure that all rows are of fixed length, OFFSET_LIMIT offsets are stored for each event, irrespective of the actual number of offsets used for the representation. Thus the space required for the flat-file representation is given by

$$Total\ Bytes(File) = (28 \times E) + (4 \times (B + P)N + E \times (8 + OFFSET\_LIMIT \times S_{\mathit{off}})) \quad (5.2)$$

Similarly, the computational complexity of the online approach depends on the average number of base-timestamp and permutation combinations searched for finding a suitable match for an event, i.e., $O((AVG(C_{search}) + 1) \times N)$, where $AVG(C_{search})$ is the average of the number of base timestamps and permutation combinations tried out for each event, i.e., $C_{search} = B_{search} \times P_{search}$.

There are two aspects involved for ordering traces online for the offset-based representation scheme. First, what combination of base timestamps and permutations are searched to find a combination that can be used to represent an event? Second, what scheme is used for reordering traces, i.e., generating new permutations?

5.3.1 Base-Timestamp and Permutation Search

In the online setting, several base timestamps and permutations are maintained in the base-timestamp and permutation caches. If the search scheme fails to find a combination of base timestamp and permutation for an event, a new base timestamp is generated and possibly a new permutation (discussed in Section 5.3.2). In this section we look at the search space of base timestamps and permutations with a *fixed interval* trace-reordering scheme where the traces are reordered after the generation of every 5 base timestamps. Thus, we always have a very recent and up-to-date trace order available and we can focus on comparing the different ways of finding a suitable combination that can be used for a new event.

There are two key aspects when searching for a suitable base-timestamp and permutation pair. The first one is the number of base timestamps and permutations cached. The second is the order in which we try out these combinations, e.g., is the most recently used base timestamp tried out with every permutation before moving on to the next base timestamp or the other way around where each permutation is tried out with all the base timestamps in the base-timestamp cache before moving on to the next permutation in cache? There are also three special cases of the two schemes described above. Thus, there are five plausible candidates for searching through the base-timestamp and permutation space. These are given as follows:

1. **All Base Timestamps with All Permutations (BTS-PERM)**: In this scheme each base timestamp in the base-timestamp cache is used in combination with all the permutations in the permutation cache for finding a suitable combination, i.e., we pick a base timestamp and try it out with all permutations before moving on to the next base timestamp.

2. **All Permutations with All Base Timestamps (PERM-BTS)**: Like the first scheme but the search order is reversed, i.e., each permutation is tried out in combination with every base timestamp in the base timestamp cache before moving on to the next permutation in the permutation cache.

3. **All Base Timestamps with the Most Recent Permutation (BTS-1)**: Similar to the first scheme but maintaining only the most recent permutation, i.e., the permutation cache size is set to 1.

4. **All Permutations with the Most Recent Base Timestamp (PERM-1)**: Differs from the previous scheme in that several permutations are maintained in cache, however, only the last generated base timestamp is maintained.

5. **Most Recent Base Timestamp with the Most Recent Permutation (1-1)**: Only the most recent base timestamp and permutation is maintained, i.e., both caches are of size 1.

Based on the results obtained by Taylor [61], we can discard the PERM-1 and 1-1 schemes because both of these schemes only maintain a single base timestamp which has been shown to be insufficient for generating an efficient event-partial-order representation. Thus, we don't entertain these schemes any further in our work as viable search schemes for online offset-based representation.

Application	Traces	Base Timestamps		Permutations	
		BTS-PERM	PERM-BTS	BTS-PERM	PERM-BTS
Life	250	61	51	13	11
ParallelLife	251	70	68	14	14
ParallelLife	501	126	124	26	25
ParallelLife	1001	247	247	50	50
Nbody	251	607	600	122	120
Nbody	501	1199	1223	240	245
Nbody	1001	2555	2554	511	511
WaveSend	251	171	181	35	37
WaveSend	501	401	435	81	87
WaveSend	1001	775	762	155	153
WaveShift	251	114	117	23	24
WaveShift	501	134	144	27	29
WaveShift	1001	346	350	70	70
Random	251	26451	26506	5291	5302

Table 5.3: Base timestamps and permutations for BTS-PERM and PERM-BTS schemes

We start by comparing the effectiveness of the BTS-PERM and PERM-BTS schemes. Note that for our analysis a cache of 256 base timestamps is maintained, which is one-fourth the maximum number of traces that we have for our target applications as per the recommendation [61]. Since a new permutation is generated every 5 base timestamps, the permutation cache size is set to 64 which is larger than one-fifth the size of the base-timestamp cache. The only difference between the BTS-PERM and the PERM-BTS schemes is the order in which base timestamps and permutation combinations are searched for finding a suitable pair for representing an event. Table 5.3 compares the total number of base timestamps and permutations generated for the two search schemes. It can be seen that the two schemes perform almost identically for all the target applications and there is only a slight difference in the numbers of base timestamps and permutations that are generated. Furthermore, the difference does not favor one scheme over the other. A

reason for the small difference between the two schemes is that for most events the first combination tried, i.e., $(1, 1)$ works and is the same for both schemes. We experimented with significantly smaller cache sizes for base timestamps and permutations and found that the BTS-PERM scheme fares significantly better than the PERM-BTS and we show in the coming sections that unlike base timestamps we do not need to maintain several permutations for online offset-based representation.

We next compare the average number of bytes required per event and the average overall search per event for the BTS-PERM and PERM-BTS schemes (shown in Table 5.4). Note that the average number of bytes per event represents the space requirement of the offset-based partial-order scheme and is calculated using Equation 5.2. Additionally, the average search is a measure of the computational cost of representing an event, since the computational complexity is given by $O((1 + AVG(C_{search})) \times N)$, where N is the number of traces (shown in column 2). The results show that the difference in the bytes per event and the average search for the two schemes is insignificant.

In Figure 5.2, we compare the average number of bytes required per event for the partial-order representation using Fidge/Mattern timestamps, Taylor's offset-based representation without trace reordering and Taylor's offset-based representation using the BTS-PERM search scheme. Note that in Figure 5.2 we only show the bytes per event for the BTS-PERM scheme and not the PERM-BTS scheme, since the space required per event is almost identical for the two schemes. Furthermore, the space also includes a fixed 28 bytes for each event. As shown in Figure 5.2, we adopt the naming convention for an online offset-based partial-order representation scheme based on the schemes used for searching the space of base timestamps and permutations and the scheme used for reordering traces, i.e., generating a new permutation. Thus, the BTS-PERM search scheme and the fixed-interval trace reordering scheme where the interval is set to 5 is referred to as BTS-PERM-FIXED-

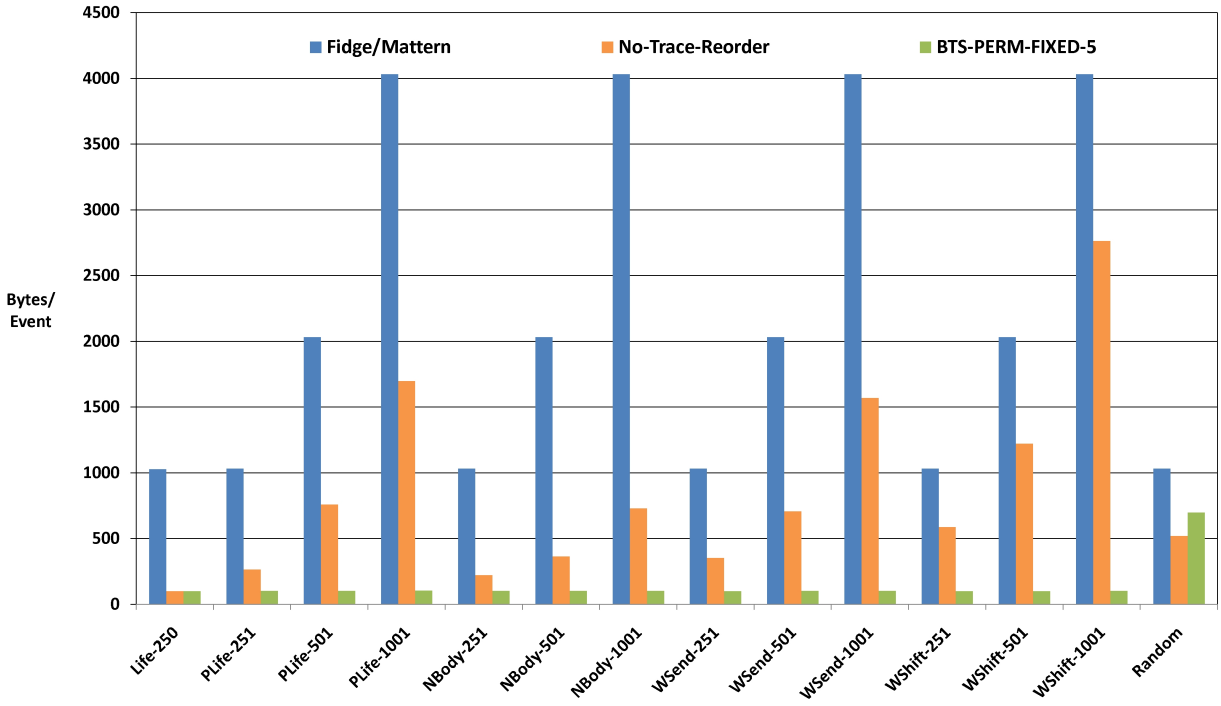

Figure 5.2: Space requirement for partial-order representation

Application	Traces	Bytes/Event		Average Search	
		BTS-PERM	PERM-BTS	BTS-PERM	PERM-BTS
Life	250	100.1	100.1	1.24	1.02
ParallelLife	251	100.8	100.8	1.23	1.21
ParallelLife	501	101.5	101.5	1.66	1.63
ParallelLife	1001	102.9	102.9	3.49	3.49
Nbody	251	100.6	100.6	6.40	6.31
Nbody	501	100.6	100.6	4.30	4.38
Nbody	1001	100.6	100.6	3.37	4.63
WaveSend	251	100.4	101.0	1.68	1.80
WaveSend	501	101.0	101.8	4.40	4.96
WaveSend	1001	101.9	101.8	5.77	5.66
WaveShift	251	100.3	100.3	1.20	1.22
WaveShift	501	100.3	100.8	1.16	1.20
WaveShift	1001	100.8	100.4	2.25	2.28
Random	251	698.5	699.5	8357	8364

Table 5.4: Bytes/event and average search for BTS-PERM and PERM-BTS schemes

5. This convention is followed throughout for the other online offset-based schemes.

Analysis of Space Efficiency

As shown in Figure 5.2, the BTS-PERM-FIXED-5 and PERM-BTS-FIXED-5 schemes achieve significant space savings compared to Fidge/Mattern timestamps. Furthermore, the space for these schemes does not increase proportionally with the number of processes. In fact, the space required for the two FIXED-5 schemes is only slightly above the theoretical minimum for online offset-based representation schemes. In an ideal scenario, only a very small number of base timestamps and permutations would be sufficient for representing all events in an application. Therefore, for an ideal case we can set the space required by the base timestamps and permutations to be 0 and compute the bytes required for each

event, we get:

$$Total\ Event\ Bytes = 28 + 8 + (AVG(Offs) \times S_{off}) = 100 \tag{5.3}$$

In a real application, however, some base timestamps and permutations will be required for the partial-order representation. Since the space required for the real target applications (i.e., excluding the synthetic Random) in our analysis is only slightly above 100 bytes, it is clear that the number of base timestamps and permutations required for efficient representation is quite small compared to the total number of events in the target application, i.e., $B << E$ and $P << E$. One reason for the space efficiency of BTS-PERM-FIXED-5 and PERM-BTS-FIXED-5 is that the traces are reordered every 5 base time-stamps. We analyze further the impact of changing the permutation-generation interval and the recency of a trace-order on the space saving achieved by the offset-based representation schemes in the next sections.

Taylor's offset-based representation essentially takes advantage of the regularity in communication of distributed and parallel applications, however, unlike previous approaches [70, 71] the offset-based schemes do not assume anything about the structure of communication patterns in applications. To demonstrate what happens if applications do not exhibit any regularity in communication, the Random MPI application was created by Taylor [61]. In Random, each process sends a message to another randomly selected process on each iteration. When using the offset-based scheme without trace reordering the average bytes required per event are almost half of the bytes required for Fidge/Mattern timestamps. Reordering traces, however, is of no use for such an application and the space required for all the generated permutations results in higher average bytes per event. Figure 5.2 shows that BTS-PERM-FIXED-5 and PERM-BTS-FIXED-5 require almost 700 bytes per event compared to the 519 bytes required for the offset-based representation scheme without

trace reordering. It is important to note that most real applications do exhibit at-least some communication regularity, exhibited either by the locality of communication or the overall communication pattern. Our results show that the event partial orders for real applications can be significantly more space-efficient when using trace-reordered variants of Taylor's offset-based representation schemes.

Analysis of Computational Complexity

As noted before, the computational complexity of trace-ordered offset-based representation schemes is proportional to the average combined search for an event and is given by $O((AVG(C_{search}) + 1) \times N)$. Table 5.4 shows the average combined search for the BTS-PERM-FIXED-5 and PERM-BTS-FIXED-5 schemes. Although the average search is significantly lower than the NO-TRACE-REORDER scheme (shown in Figure 5.3), the search is orders of magnitude higher than the NO-TRACE-REORDER scheme for Random (shown in Table 5.2 and Table 5.4). The main reason for such a high average search is that for the large number of events for which no base-timestamp and permutation combination works, all the possible base-timestamp and permutation combinations are tried out. We explore the combined average search further in the next section.

Permutation Cache Size

There are two factors that contribute to the space efficiency of the BTS-PERM-FIXED-5 and PERM-BTS-FIXED-5 schemes. First, the frequency of reordering traces, i.e., the FIXED-5 trace-reordering scheme where a permutation is generated every 5 base timestamps and secondly, the number of permutations in the permutation cache, set to 64 for the results presented here. We explore the interval for reordering traces and the computa-

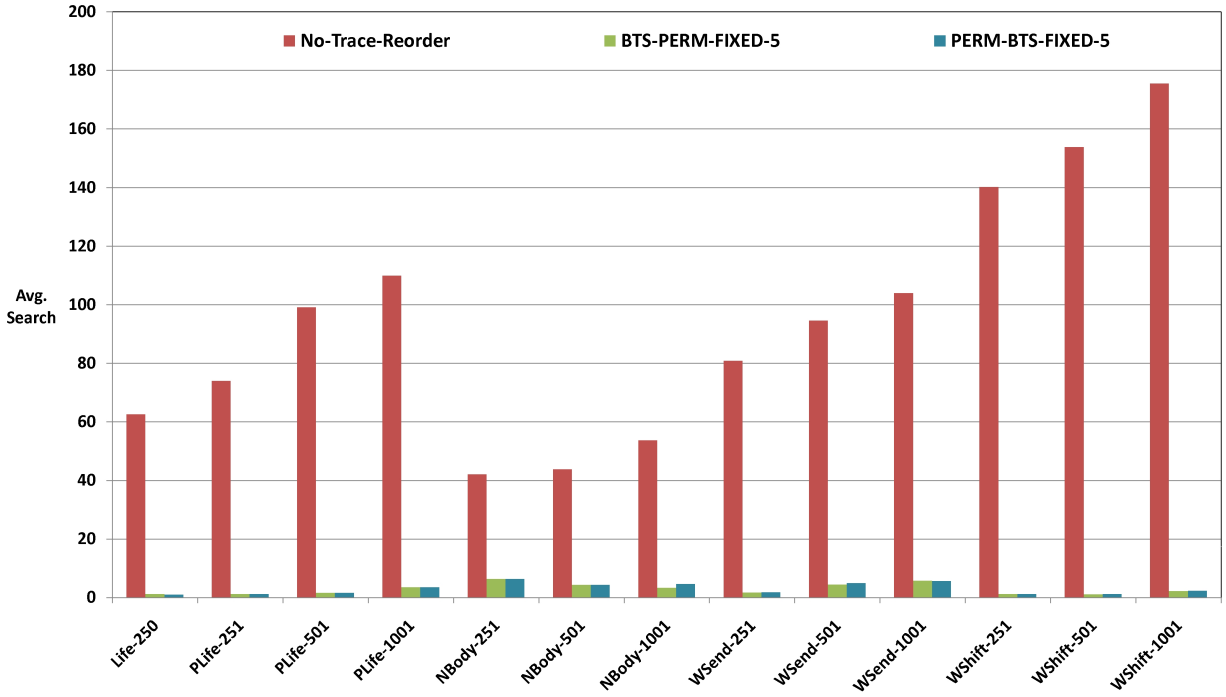

Figure 5.3: Average search for base timestamp and permutation combination

Application	Events	Permutation Search (%)					
		1	2	3	4	5	6-64
Life-250	502500	97.98	1.30	0.20	0.10	0.40	0.02
PLife-251	101502	99.90	0.03	0.01	0.01	0.01	0.05
PLife-501	203002	99.91	0.02	0.00	0.00	0.00	0.06
PLife-1001	406002	99.92	0.02	0.00	0.00	0.00	0.06
Nbody-251	1256502	99.86	0.07	0.00	0.00	0.00	0.06
Nbody-501	5013002	99.89	0.06	0.01	0.01	0.00	0.04
Nbody-1001	20026002	98.38	0.47	0.16	0.11	0.10	0.78
WSend-251	500248	99.85	0.08	0.02	0.00	0.00	0.04
WSend-501	1002498	99.84	0.09	0.02	0.00	0.00	0.05
WSend-1001	2006998	99.84	0.08	0.02	0.00	0.00	0.05
WShift-251	500248	99.97	0.01	0.00	0.00	0.00	0.02
WShift-501	1002498	99.98	0.00	0.00	0.00	0.00	0.01
WShift-1001	2006998	99.98	0.00	0.00	0.00	0.00	0.02
Random	53248	27.52	1.74	0.28	0.31	0.25	69.9

Table 5.5: Distribution of permutations searched for BTS-PERM-FIXED-5

tional complexity of that operation in Section 5.3.2. In this section, we analyze the impact of the number of permutations maintained in the permutation cache on the space and computational requirement of the online offset-based schemes. A similar analysis for the base-timestamp cache was done by Taylor [61] where the size of the base-timestamp cache is varied and its impact on the space-efficiency of the offline offset-based schemes is evaluated. The results showed that a very small cache can result in a lot of base timestamps being generated and therefore, an adequately sized base timestamp cache, e.g., one-fourth the total number of traces, must be maintained for most real applications.

Table 5.5 shows the distribution of permutations examined to find a successful base timestamp and permutation combination for the offset-based representation for all the target applications when the scheme used is BTS-PERM-FIXED-5. We obtained very similar results for the PERM-BTS-FIXED-5 scheme. As shown in Table 5.5, for all real

applications the most recently used permutation, by a very large margin, was most likely to be useful in representing a new event. Based on these results it is natural to assume that only the single most recent trace order should be considered a viable candidate for a new event. One must however, note the distinction between the most recently used and the most recently generated permutation and the above results only show that the permutation that was successfully used most recently is likely to be the most likely successful candidate for the next event. Taylor [61] showed that the above holds true for base timestamps as well, however, for base timestamps there is a disparity between the most recently used base timestamp and the most recently generated base timestamp. For base timestamps a non-recent base timestamp can be useful for a new event and brought to the "front" of the most recently used base-timestamp vector.

A distinction between base timestamp and permutation is that a new timestamp only captures the subset of the current communication in the application and depends on the communication locality. On the other hand, trace orders are global and a new trace order moves all traces that communicate with each other close to each other and in a way captures the global communication pattern of an application. Therefore, a number of base timestamps may represent a subset of the current state of communication within an application, whereas we do not need multiple trace orders to represent the overall communication pattern in an application. This distinction between base timestamps and permutations and the average permutation search for real applications (Table 5.5) motivated an investigation of maintaining only the most recently generated permutation. The resulting scheme is BTS-1, where the permutation cache only maintains the most recently generated permutation. Yet another motivation for the BTS-1 scheme is that the combined average search for the "cache miss" cases is two orders of magnitude smaller than the BTS-PERM and PERM-BTS scheme since only $B \times 1$ combinations are tried instead of the $B \times P$ combinations

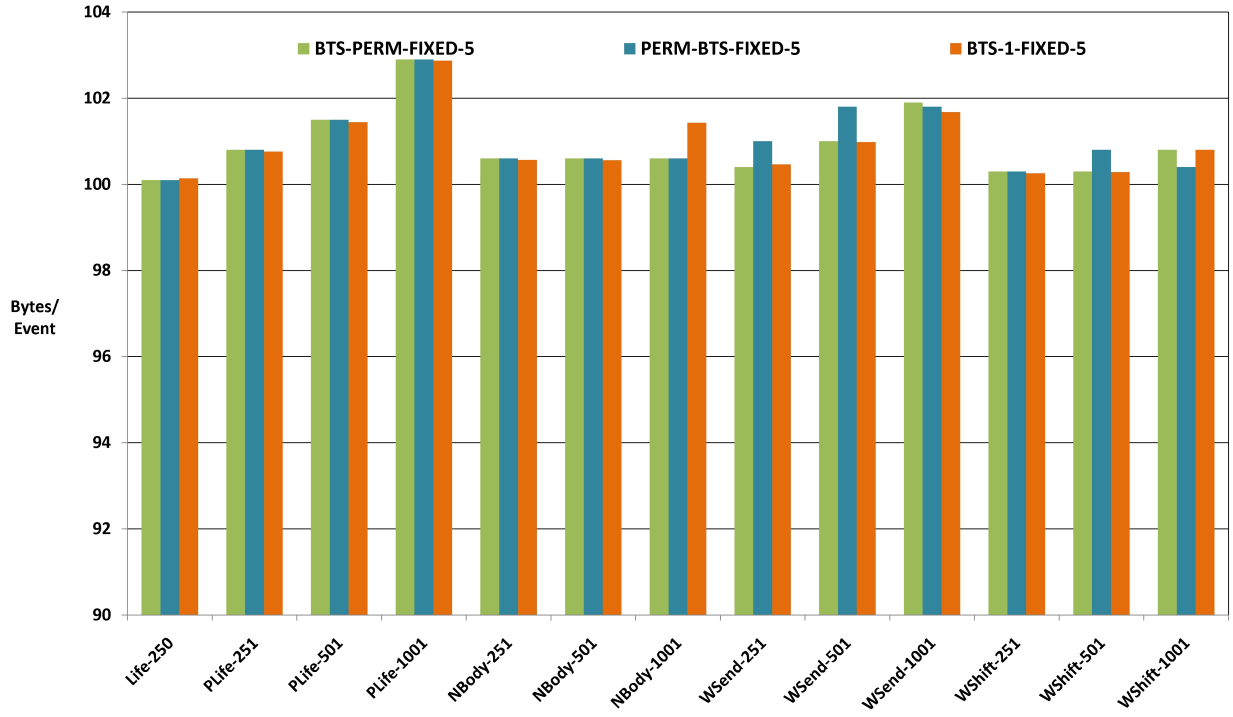

Figure 5.4: Space comparison of BTS-PERM, PERM-BTS and BTS-1

for all base timestamps and permutations.

We next compare the space required per event for the three online offset-based schemes, i.e., BTS-PERM-FIXED-5, PERM-BTS-FIXED-5 and BTS-1-FIXED-5, shown in Figure 5.4. The results show that our hypothesis that only the most recently generated permutation is a viable candidate for representing a new event does in fact hold, as we see practically no difference in the space required for representing events. The only exception is Random (not shown for scale) where the average bytes required per event using BTS-1-FIXED-5 is 817.2 compared to 698.5 and 699.7 bytes/event for BTS-PERM-FIXED-5 and PERM-BTS-FIXED-5. As expected, we see significant gains in average search when only maintaining the last permutation. Figure 5.5 shows the combined average search for MPI target applications. Note that the theoretical minimum search for offset based schemes is 1,

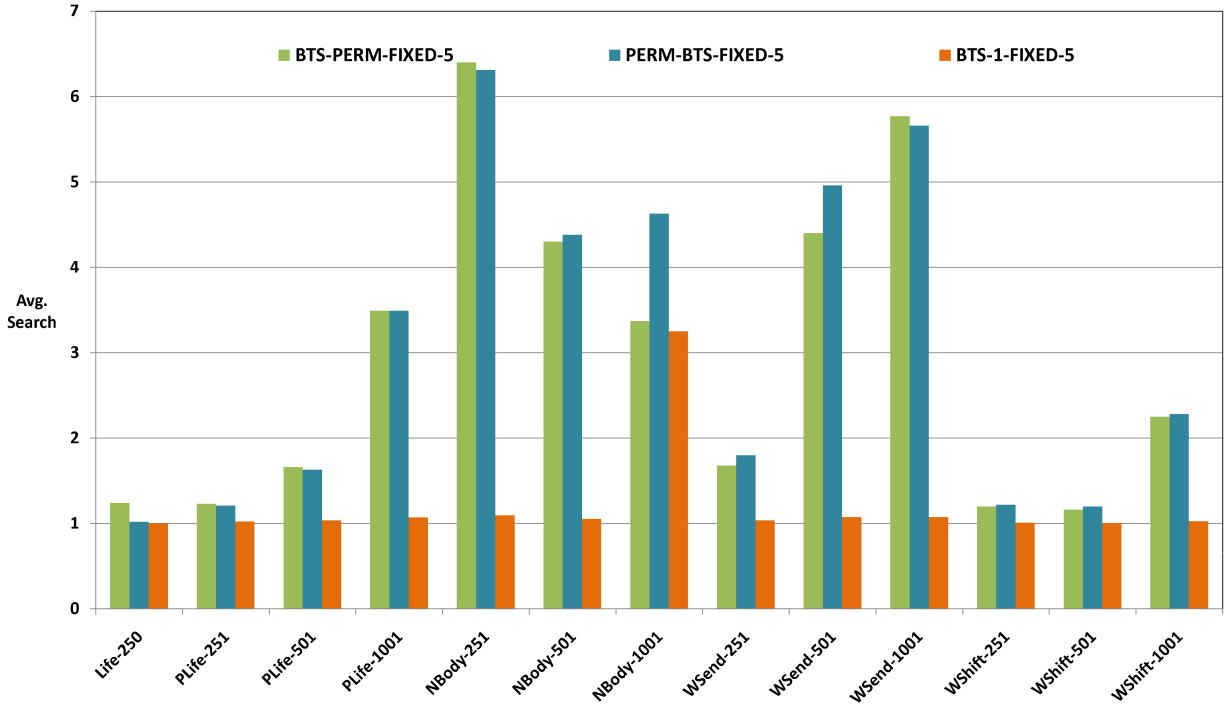

Figure 5.5: Average combination search for BTS-PERM, PERM-BTS and BTS-1

where only one base timestamp is tried out in combination with one permutation. As shown in Figure 5.5 the combined search is very small in most cases, thus significantly reducing the computational complexity of generating a partial-order representation as compared to the BTS-PERM and PERM-BTS schemes. Again, we intentionally omitted Random so as not to distort the scale of Figure 5.5. Table 5.6 summarizes the average search for the three schemes for all MPI applications including Random. Although the average search is significantly higher than for real applications, it is an order of magnitude smaller than the BTS-PERM and PERM-BTS schemes where the failed searches required going through all the possible base-timestamp and permutation combinations.

Although the space required for the three schemes is almost identical and significantly better than the offset-based schemes without ordering traces, the BTS-1 scheme is signifi-

Application	Traces	Average Search		
		BTS-PERM	PERM-BTS	BTS-1
Life	250	1.24	1.02	1.003
ParallelLife	251	1.23	1.21	1.022
ParallelLife	501	1.66	1.63	1.037
ParallelLife	1001	3.49	3.49	1.072
Nbody	251	6.40	6.31	1.095
Nbody	501	4.30	4.38	1.054
Nbody	1001	3.37	4.63	3.251
WaveSend	251	1.68	1.80	1.04
WaveSend	501	4.40	4.96	1.074
WaveSend	1001	5.77	5.66	1.075
WaveShift	251	1.20	1.22	1.011
WaveShift	501	1.16	1.20	1.007
WaveShift	1001	2.25	2.28	1.026
Random	251	8357	8364	152.296

Table 5.6: Average search for BTS-PERM, PERM-BTS and BTS-1 schemes

cantly more computationally efficient than BTS-PERM and PERM-BTS schemes because of the lower average base-timestamp and permutation search.

The Cost of Reordering Traces

In our analysis so far, all schemes work with the FIXED-5 trace-reordering scheme where a new permutation is generated after every 5 base timestamps. The number of permutations generated, therefore, is simply one-fifth the number of base timestamps (shown in Table 5.3). Although, the amortized cost over all events of generating base timestamps and permutations is small, since $B << E$ and $P << E$, the computational cost of generating a new permutation is significantly more than the computational cost of generating a new base timestamp. Generating a new base timestamp is trivial and equates to storing a copy of the already generated N-element Fidge/Mattern vector timestamp. On the other hand,

the cost of generating a new permutation is $O(N^3)$, where N is the number of traces in the target application. To further reduce the overall computational cost of the offset-based schemes we need to reduce the total number of trace-reordering operations. In the following section we look at the interval for reordering traces with the goal of reducing the number of permutations without significantly compromising the space effectiveness of the online offset-based schemes.

5.3.2 Generating Permutations

There are two aspects for a permutation-generation scheme, firstly, the trigger for reordering traces, i.e., can a new permutation be generated at any time or only when a new event is received or when a new base timestamp is generated. Secondly, the trace-reorder interval and the criteria for reordering traces.

Trigger for Reordering Traces

In principle, we could generate a new permutation at any point, i.e., at any time during the execution of the target application, but as long as no new base timestamp is generated, there is no reason to reorder traces. Therefore, a reasonable trigger for generating a new permutation is when all the existing base timestamps and permutations do not work for an event and a new base timestamp must be generated. By generating a new permutation only after a new base timestamp is generated we avoid incurring the additional computational cost associated with the generation of superfluous permutations.

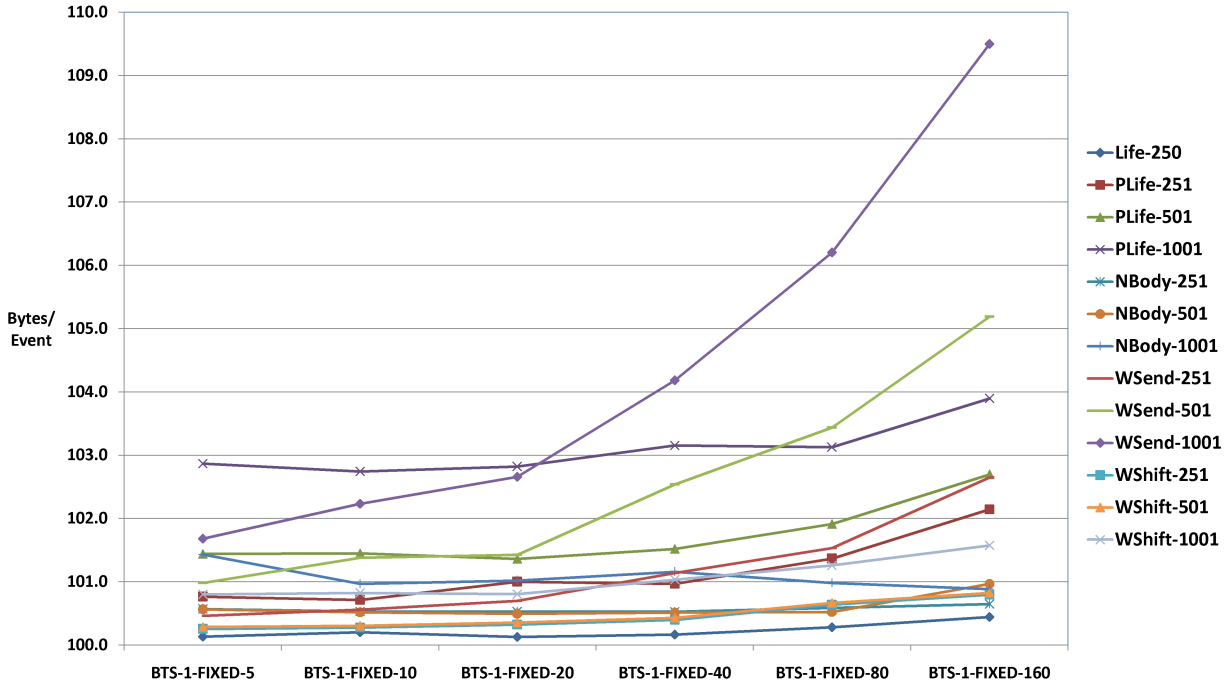

Figure 5.6: Bytes/event for BTS-1 with trace-reorder interval from 5 to 160

Trace-Reorder Interval

We start by looking at the effectiveness of the offset-based schemes when the trace-reorder interval is varied, specifically the space and the computational efficiency of the offset-based schemes. Note that, although the trace-reorder interval is fixed relative to the base timestamps, the number of events represented in each interval varies. Figure 5.6 shows the impact on the space, i.e., bytes per event required for the BTS-1 search scheme when the trace-reorder interval is varied from 5 to 160. The changes in the bytes required per event depend on the number of additional base timestamps generated when the trace-reorder interval is varied and the number of events represented per base timestamp. The number of events represented per base timestamp is dependent on the application and for example, varies from under 2 for Random to around 9000 for MPI Life-250 when using the

Application	Events	Trace-Reorder Interval					
		5	10	20	40	80	160
Life-250	502500	100.1	100.2	100.1	100.2	100.3	100.4
PLife-251	101502	100.8	100.7	101.0	101.0	101.4	102.1
PLife-501	203002	101.4	101.5	101.4	101.5	101.9	102.7
PLife-1001	406002	102.9	102.7	102.8	103.2	103.1	103.9
Nbody-251	1256502	100.6	100.5	100.5	100.5	100.6	100.6
Nbody-501	5013002	100.6	100.5	100.5	100.5	100.5	101.0
Nbody-1001	20026002	101.4	101.0	101.0	101.2	101.0	100.9
WSend-251	500248	100.5	100.6	100.7	101.1	101.5	102.7
WSend-501	1002498	101.0	101.4	101.4	102.5	103.4	105.2
WSend-1001	2006998	101.7	102.2	102.7	104.2	106.2	109.5
WShift-251	500248	100.3	100.3	100.3	100.4	100.6	100.8
WShift-501	1002498	100.3	100.3	100.4	100.4	100.7	100.8
WShift-1001	2006998	100.8	100.8	100.8	101.0	101.3	101.6
Random	53248	817.2	746.1	702.9	674.9	639.5	592.8

Table 5.7: Bytes/event for BTS-1 with trace-reorder intervals from 5 to 160

BTS-1-FIXED-5 scheme for partial-order representation. Figure 5.6 shows that increasing the trace-reorder interval does not always result in a corresponding increase in the space required for representation. NBody-1001, for example, sees little change in the space required when the trace-reorder interval increases from 5 to 160. On the other hand, for WaveSend-1001, the bytes required per event increases uniformly with the size of the trace-reorder interval. Furthermore, other applications such as Life-250, require more bytes per event when the trace-reorder interval is greater than 40. Table 5.7 summarizes the results depicted in Figure 5.6 and also includes Random (not shown in Figure 5.6). For it, the average number of bytes required per event uniformly decreases, which is to be expected since reordering traces is of no help and space is saved because fewer permutations are generated.

Since the changes in the space required for representation is dependent on the number

Application	Events	Trace-Reorder Interval					
		5	**10**	**20**	**40**	**80**	**160**
Life-250	502500	56	91	61	81	140	221
PLife-251	101502	64	65	96	95	136	215
PLife-501	203002	121	133	131	150	191	271
PLife-1001	406002	242	252	272	312	313	392
Nbody-251	1256502	591	607	627	645	726	804
Nbody-501	5013002	1173	1182	1174	1256	1292	2408
Nbody-1001	20026002	5969	4400	4846	5642	4854	4388
WSend-251	500248	192	253	331	553	753	1312
WSend-501	1002498	408	628	679	1238	1699	2579
WSend-1001	2006998	701	1017	1270	2045	3071	4733
WShift-251	500248	105	125	154	194	314	394
WShift-501	1002498	119	139	169	209	329	409
WShift-1001	2006998	334	374	384	504	624	784
Random	53248	31699	31153	30453	29748	28258	25975

Table 5.8: Base timestamps for BTS-1 with trace-reorder intervals from 5 to 160

of base timestamps we next look at the impact of varying the trace-reorder interval on the number of base timestamps. Table 5.8 shows the number of base timestamps generated for the BTS-1 scheme as the trace-reorder interval is increased from 5 to 160. Note that the variation in the number of base timestamps is non-uniform across different applications. These results are represented in Figure 5.7 using a logarithmic scale for the number of base timestamps. For NBody-501 and ParallellLife-1001, we do not see a noticeable increase in the number of base timestamps generated up to a trace-reorder interval of 80, whereas for WaveSend, the number of base timestamps increases as the trace-reorder interval increases.

The results above show that, although BTS-1-FIXED-5 is generally the most space-efficient scheme, not all applications require such aggressive trace-reordering. We next analyze the impact on the computational requirement of the representation process as the

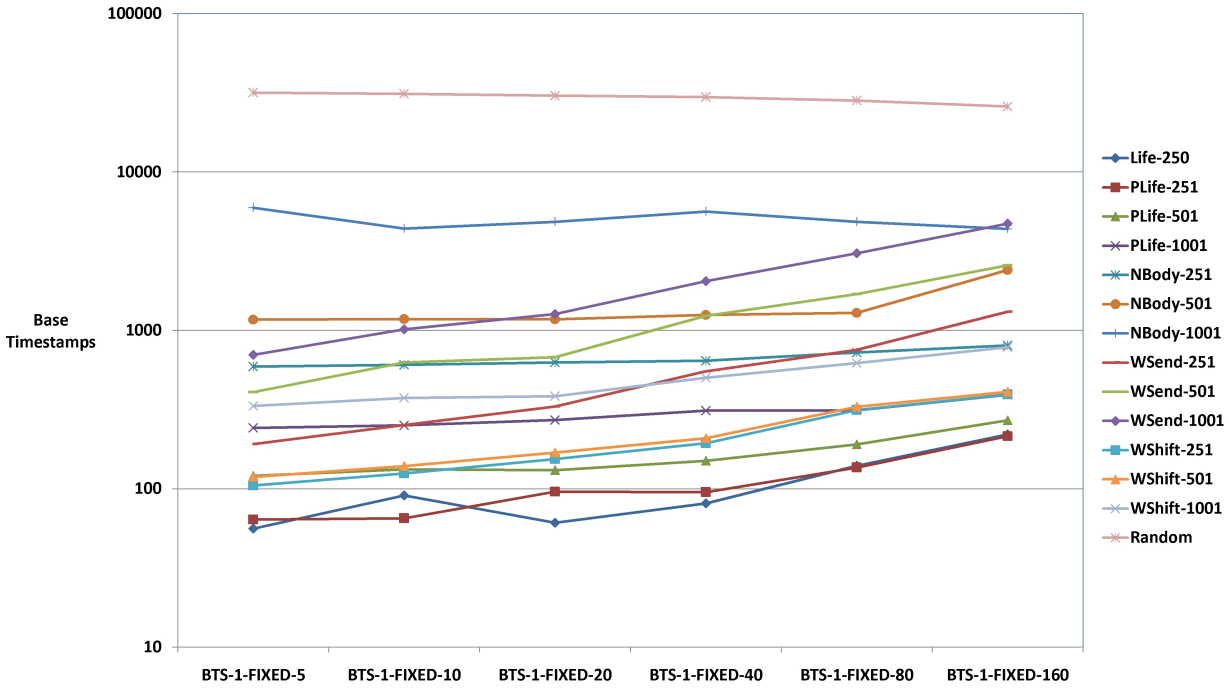

Figure 5.7: Base timestamps for BTS-1 with trace-reorder interval from 5 to 160

Application	Events	Trace-Reorder Interval					
		5	**10**	**20**	**40**	**80**	**160**
Life-250	502500	12	10	4	3	2	2
PLife-251	101502	13	7	5	3	2	2
PLife-501	203002	25	14	7	4	3	2
PLife-1001	406002	49	26	14	8	4	3
Nbody-251	1256502	119	61	32	17	10	6
Nbody-501	5013002	235	119	59	32	17	16
Nbody-1001	20026002	1194	440	243	142	61	28
WSend-251	500248	39	26	17	14	10	9
WSend-501	1002498	82	63	34	31	22	17
WSend-1001	2006998	141	102	64	52	39	30
WShift-251	500248	21	13	8	5	4	3
WShift-501	1002498	24	14	9	6	5	3
WShift-1001	2006998	67	38	20	13	8	5
Random	53248	6340	3116	1523	744	354	163

Table 5.9: Permutations generated for BTS-1 with trace-reorder intervals from 5 to 160

trace-reorder interval is varied. There are two computational costs of interest, first, the actual number of trace-reorder operations since they are expensive, i.e., $O(N^3)$ where N is the number of traces in the application. The second is the impact on the average combined search, since reducing the number of trace-reorder operations would be counterproductive if it resulted in significant increases in the average search for representing an event. Table 5.9 shows the number of trace-reorder operations, i.e, the number of permutations generated as the trace-reorder interval is varied from 5 to 160. The results align with the results in Table 5.8 in that doubling the trace-reorder interval does not result in a corresponding doubling of the number of base timestamps for most applications. Figure 5.8 depicts this phenomenon. The applications for which there is a sharp decrease in the number of permutations as the trace-reorder interval is increased are the ones where increasing the interval does not result in a corresponding increase in the number of base timestamps and

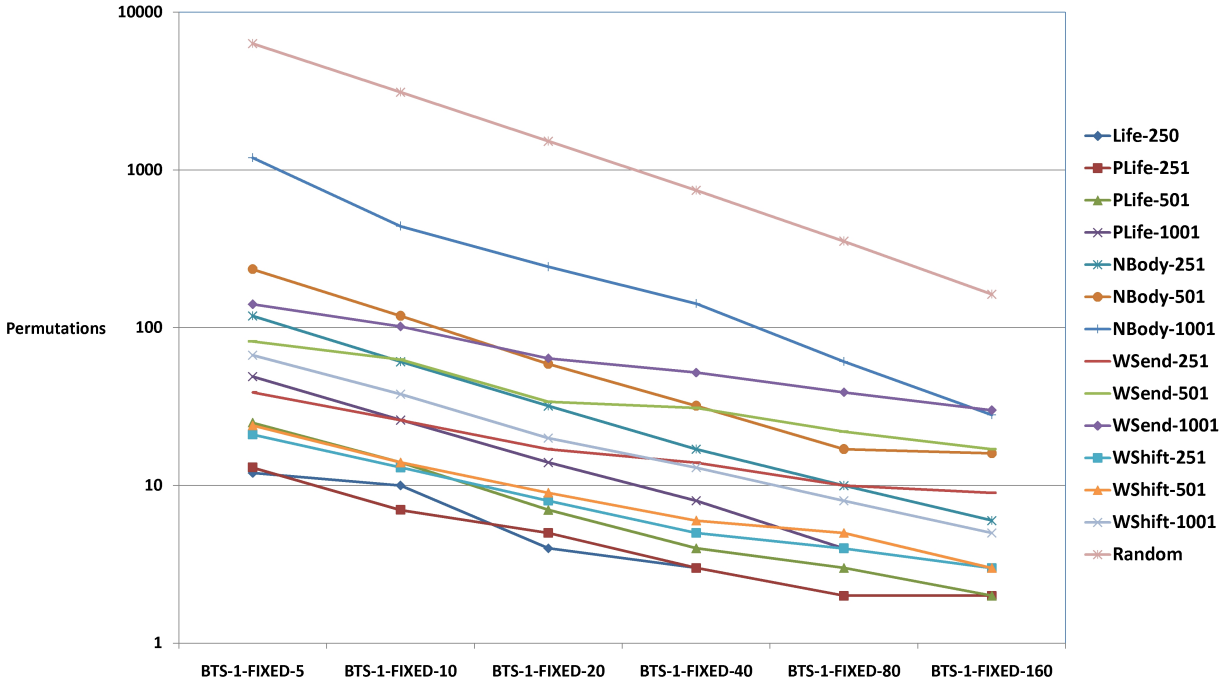

Figure 5.8: Permutations for BTS-1 with trace-reorder interval from 5 to 160

thus the overall space efficiency remains the same.

Lastly, we look at the impact of increasing the trace-reorder interval on the average combined search for finding a base timestamp and permutation combination for representing an event. Table 5.10 shows the average combined search for the BTS-1 scheme when the trace interval is successively doubled from 5 to 160. The results show that too large a trace-reordering interval can result in significant increase in the average search, for example, the average combined search for Life-250 jumps from almost 1 to over 4 when the trace-order interval is increased from 40 to 80.

Application	Events	Trace-Reorder Interval					
		5	**10**	**20**	**40**	**80**	**160**
Life-250	502500	1.003	1.008	1.004	1.008	4.228	4.251
PLife-251	101502	1.022	1.021	1.058	1.049	1.108	1.362
PLife-501	203002	1.037	1.045	1.042	1.058	1.097	1.197
PLife-1001	406002	1.072	1.078	1.091	1.131	1.119	1.183
Nbody-251	1256502	1.095	1.098	1.104	1.111	1.155	1.242
Nbody-501	5013002	1.054	1.054	1.054	1.060	1.062	2.619
Nbody-1001	20026002	3.251	2.413	3.283	4.311	4.105	3.510
WSend-251	500248	1.039	1.069	1.117	1.253	1.362	2.163
WSend-501	1002498	1.074	1.136	1.158	1.347	1.464	1.724
WSend-1001	2006998	1.075	1.119	1.158	1.277	1.420	1.677
WShift-251	500248	1.011	1.016	1.025	1.042	2.033	1.219
WShift-501	1002498	1.007	1.010	1.015	1.023	1.061	1.093
WShift-1001	2006998	1.026	1.032	1.033	1.050	1.091	1.263
Random	53248	152.296	149.697	146.381	143.253	136.974	128.678

Table 5.10: Average search for BTS-1 with trace-reorder interval from 5 to 160

Dynamic Interval Selection

The analysis in the previous section shows that for various applications different trace-reordering intervals can be used to significantly reduce the total number of permutations that are generated without significantly compromising the space-efficiency of the partial order representation. Clearly the results show that an interval can be selected statically that works best for a particular application, however, such a trace-reordering scheme would not be suitable in an online setting. Furthermore, applications vary significantly with respect to a number of parameters, e.g., the number of base timestamps generated per event and thus any pre-defined threshold would not work well for all applications. We start by giving examples of the number of base timestamps that are generated for two applications as a function of the number of events in these applications.

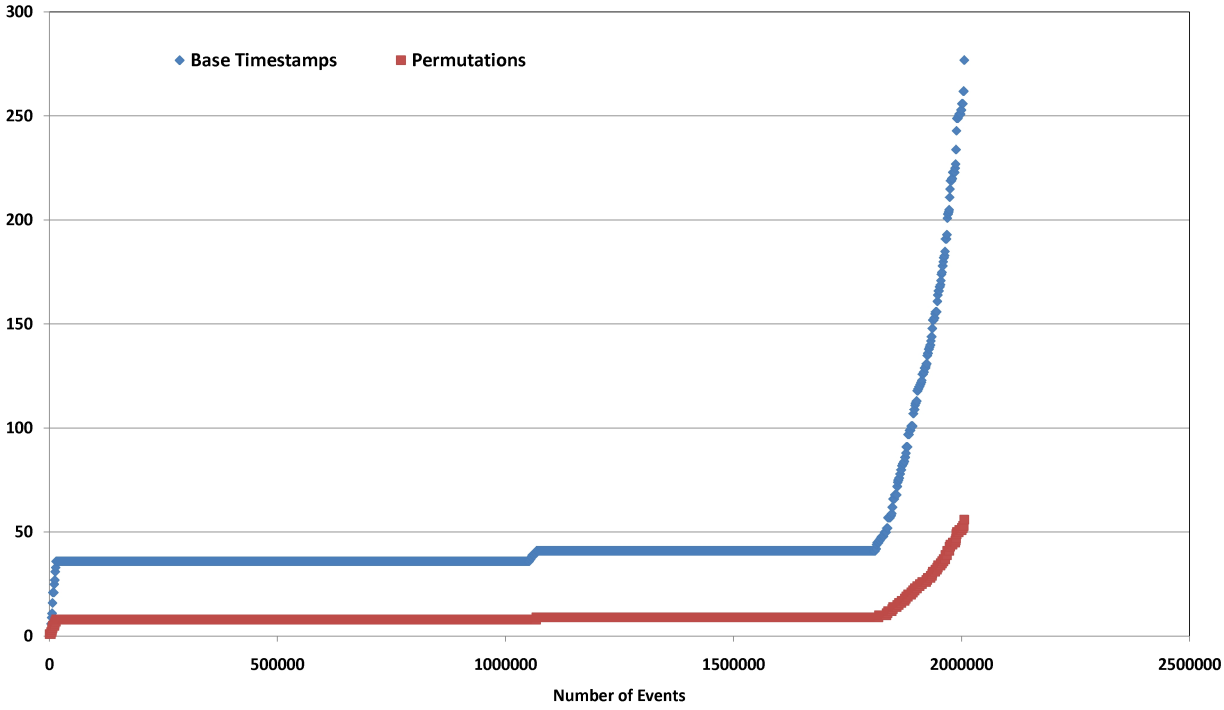

Figure 5.9: Base timestamps and permutations (FIXED-5) for WaveShift-1001

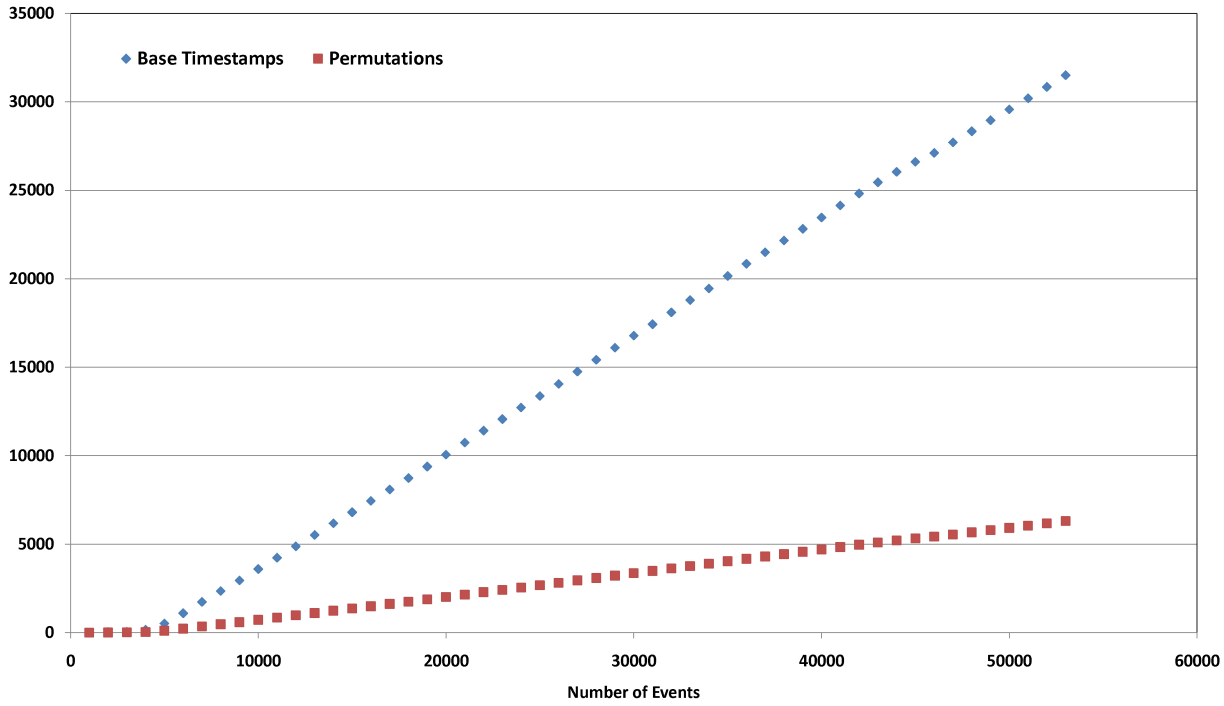

Figure 5.10: Base timestamps and permutations (FIXED-5) for Random-251

Figure 5.9 shows the number of base timestamps and permutations generated as a function of the number of events in the WaveShift application with 1001 processes. Numerous base timestamps and consequently numerous permutations are generated when the application starts and also when processes terminate near the end. We observed this 'flurry' of base timestamps and permutations for many other applications as well where during certain periods, the communication pattern of the application underwent significant changes. Such changes often result in the generation of a large number of base timestamps. Similarly, Figure 5.10 shows the base timestamps generated for the Random-251. Unlike most applications, Random does not benefit from trace reordering, therefore, the number of base timestamps increases uniformly with the number of represented events.

The observations made about the distributions of generated base timestamps as a func-

tion of represented events motivated the development of a *dynamic interval scheme* that we propose in our work. Real applications go through long uniform phases that require generation of few new base timestamps and some phases in which a flurry of activity occurs such as when the application starts and new processes are created or during another phase where a major shift occurs in the communication pattern of the application. Since during a calm phase, the number of base timestamps generated is small and the number of events represented per base timestamp is very large, the trace reordering interval should ideally be small. On the other hand, when there is a flurry of activity, frequent trace reordering would be of little help until the application settles down, such as for WaveShift when the application starts (shown in Figure 5.9). Therefore, for a dynamic trace-reordering scheme we ideally want the trace-reorder interval to be large during such periods so as not to generate superfluous permutations.

In the dynamic inverse interval scheme, at the end of an interval, i.e., when a new trace-order is about to be generated, the average number of events represented per base timestamp in that interval is compared with the average number of events represented during the last interval. We refer to the average number of events represented per base timestamp as the *timestamp effectiveness* metric for a particular interval. If for the current interval the timestamp-effectiveness is more than the previous interval, the next interval is selected to be half of the current interval or the minimum interval specified. Similarly, if the timestamp effectiveness is less than the last interval, the length of the next trace reorder interval is doubled. It might seem counter-intuitive to increase the interval when the offset-based scheme is less effective, however, as noted before, our analysis shows that reordering traces does not help when the application is undergoing a major flux, e.g., when it starts or terminates. Similarly, it does not hurt to reduce the interval when the offset-based scheme is effective, because by definition, more events are being represented per base

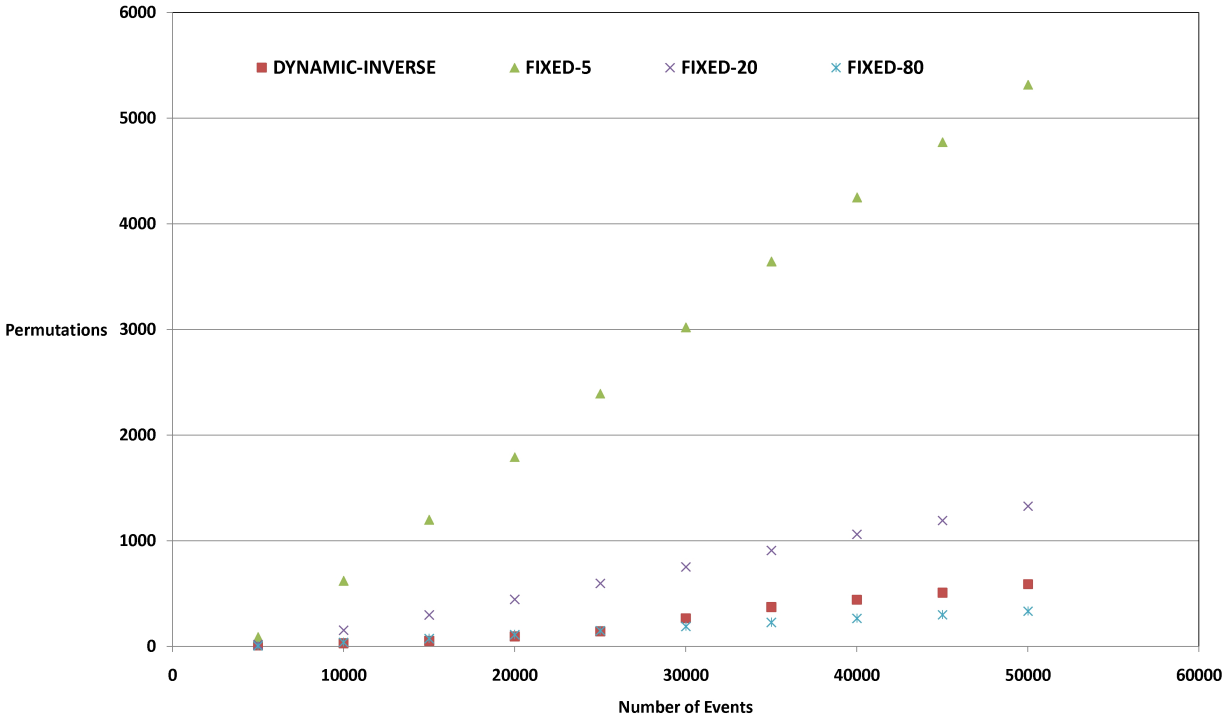

Figure 5.11: Permutations for Random-251 using DYNAMIC-INVERSE

timestamp and thus fewer base timestamps are generated that can in turn trigger a trace reordering.

Figure 5.11 shows the permutations generated for a run of Random-251 using the dynamic inverse interval scheme where the initial, minimum, and maximum interval is set to 5, 5, and 160. We think that any interval lower than 5 would be too small even for very small applications and most real applications do not require an interval as large as 160. Furthermore, the initial interval is set to the minimum interval so as not to incur overhead for "smaller" applications where any higher interval would be too large and would therefore, result in the generation of a significant number of additional base timestamps. As can be seen, the average trace-reorder interval for Random is approximately 45.

Application	BTS	Permutations	Avg. Interval	Search	Bytes/Event
Life-250	132	11	12	1.13	100.3
PLife-251	65	8	8.1	1.02	100.7
PLife-501	117	10	11.7	1.03	101.3
PLife-1001	244	16	15.3	1.07	102.6
Nbody-251	597	43	13.9	1.10	100.5
Nbody-501	1183	46	25.7	1.05	100.5
Nbody-1001	4502	109	41.3	3.34	100.9
WSend-251	405	26	15.6	1.38	100.9
WSend-501	1172	42	27.9	1.36	102.4
WSend-1001	1848	81	22.8	1.27	103.8
WShift-251	126	10	12.6	1.02	100.3
WShift-501	157	11	14.3	1.01	100.3
WShift-1001	413	14	29.5	1.04	100.9
Random	28017	479	58.5	136.80	637.0

Table 5.11: Statistics for all applications with BTS-1-DYNAMIC-INVERSE-5-5-160

Table 5.11 summarizes the results obtained for all applications when the BTS-1-DYNAMIC-INVERSE-5-5-160 scheme is used for search and trace reordering. Note that the average trace-reorder interval varies for different applications, from as low as 8 for life-250 to as high as 60 for Random. It makes sense that for an application like Random, which does not benefit from any trace reordering, the overall average trace reorder interval is very high. The converse also holds for applications that do benefit from frequent trace reordering. The average search and the bytes per event are slightly higher than when FIXED-5 trace reordering scheme is used, however, we also see a significant reduction in the number of trace-reordering operations. Although we can pick an ideal static interval for an application, such a scheme would be of little use in an online setting. A significant strength of the proposed dynamic inverse scheme is that the scheme adapts in a dynamic manner based on the effectiveness of the offset-based representations, while at the same time significantly reducing the number of trace-reordering operations.

In the next section we analyze the run-time variations in the computational and space costs of representing events using the online offset-based partial-order-representation scheme. We then compare the effectiveness of the online partial order representation scheme (BTS-1-DYNAMIC-INVERSE-5-5-160) with the offline variant presented in the original work [61]. Lastly, we look at the space required for storing the representation in a relational database as opposed to a flat file.

5.4 Further Analysis

5.4.1 Run-Time Variations and Confidence Intervals

For most real applications where a large number of events are generated the monitoring application may not receive the events in the same order as they are generated by the target application. For example, when using POET for monitoring, the target application opens a number of streams to the POET server, each stream is used to send the events generated on a number of traces. Note that event n on trace i would always be represented by POET before an event $n+1$ on trace i, however, event n or trace i may be represented after event m on trace j even if event n might have been emitted before event m by the target application. As a direct consequence, the partial order constructed would always be the same for multiple replays of the same application, however, the base timestamps and permutations generated may be different.

In this section we analyze the overall variations in the number of base timestamps and permutations generated using the BTS-1-DYNAMIC-INVERSE-5-5-160 scheme for 5 runs of each target application. Figure 5.12 and Figure 5.13 show the average number of base timestamps and permutations generated and the 95% confidence interval for the 5

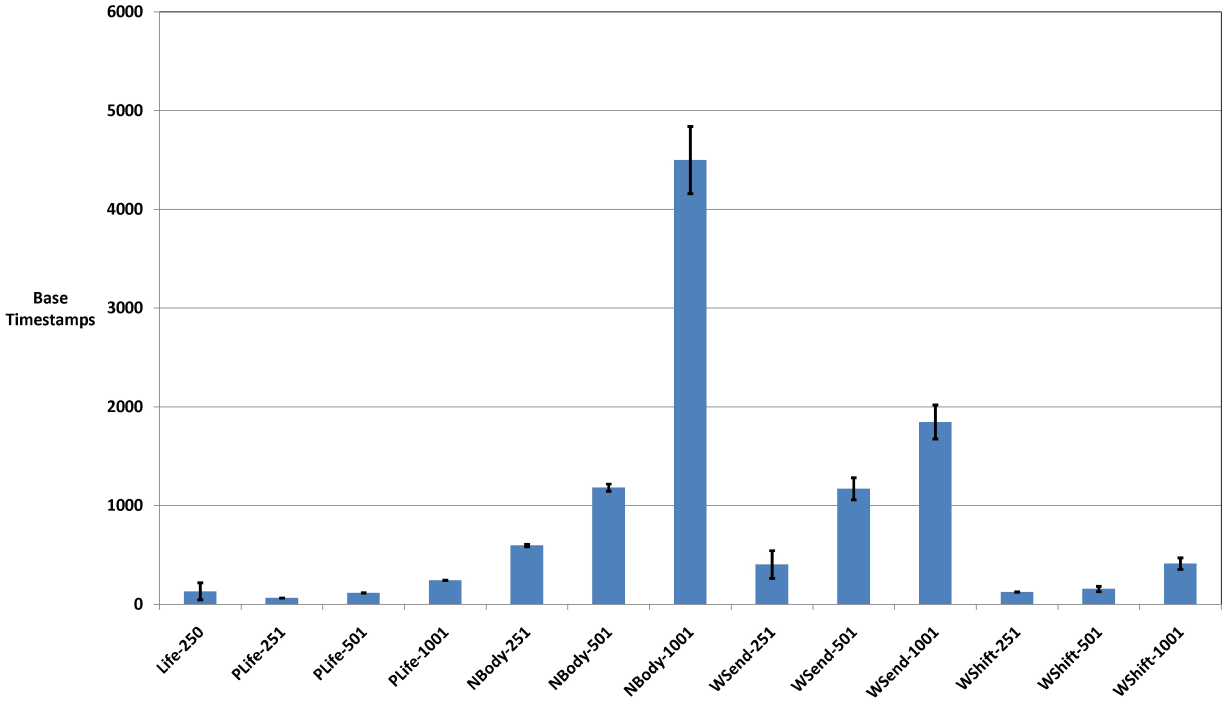

Figure 5.12: Base timestamps using DYNAMIC-INVERSE-5-5-160

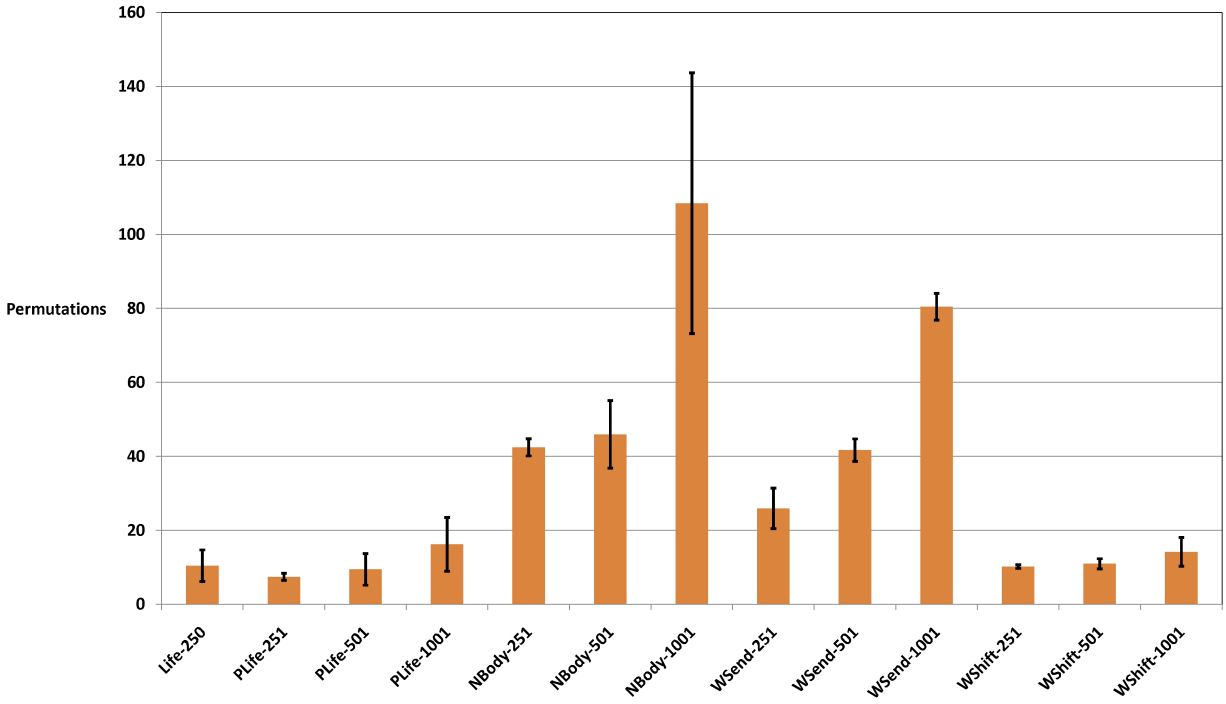

Figure 5.13: Permutations using DYNAMIC-INVERSE-5-5-160

runs. The confidence intervals are calculated with the assumptions that the number of base timestamps and permutations generated follow a normal distribution with unknown standard deviation. The results show that there are more variations in the total number of permutations generated than for the base timestamps, however, one must note that the number of permutations generated is relatively small compared to the number of base timestamps and small fluctuations result in larger relative variations.

Similarly, Table 5.12 summarizes the overall variations in the space required for representation and the average search for five runs of all target applications. The results show that the overall variation in search as well as space is relatively small and the the average 95% confidence intervals for search and bytes per event are approximately within $\pm 5\%$ and $\pm 1\%$ of the average value.

Application	Avg. Search	Bytes/Event
Life-250	$1.13 \pm .23$	$100.3 \pm .2$
PLife-251	$1.02 \pm .00$	$100.7 \pm .0$
PLife-501	$1.03 \pm .00$	$101.3 \pm .0$
PLife-1001	$1.07 \pm .00$	$102.6 \pm .1$
Nbody-251	$1.10 \pm .00$	$100.5 \pm .0$
Nbody-501	$1.05 \pm .00$	$100.5 \pm .0$
Nbody-1001	$3.34 \pm .36$	$100.9 \pm .1$
WSend-251	$1.38 \pm .27$	$100.9 \pm .3$
WSend-501	$1.36 \pm .16$	$102.4 \pm .2$
WSend-1001	$1.27 \pm .07$	$103.8 \pm .3$
WShift-251	$1.02 \pm .00$	$100.3 \pm .0$
WShift-501	$1.01 \pm .01$	$100.3 \pm .1$
WShift-1001	$1.04 \pm .01$	$100.9 \pm .1$
Random	$136.80 \pm .21$	637.0 ± 8.0
	$\approx \pm 5\%$	$\approx \pm 1\%$

Table 5.12: 95% confidence intervals for search and bytes/event

5.4.2 Comparison with the Offline Offset-Based Scheme

In this section we compare the online offset-based representation scheme, i.e., BTS-1-DYNAMIC-INVERSE-5-5-160 with the offline offset-based representation scheme presented in the original work [61]. For online and offline schemes the size of the base timestamp cache is set to 256 and the OFFSET_LIMIT is set to 4. Note that for the offline scheme, once all events are seen by POET, a single permutation is generated and is used for the representation phase.

As shown in Table 5.13, except for Random, the difference between the space required for the online and offline schemes is consistently slightly over 4 bytes. For the online scheme these 4 bytes are contributed by the permutation pointer maintained for each event. Aside from this overhead, the additional difference in space, which is very small, is contributed by

Application	Avg. Search		Bytes/Event	
	Offline	Online	Offline	Online
Life-250	1.00	1.13	96.0	100.3
PLife-251	1.02	1.02	96.5	100.7
PLife-501	1.03	1.03	97.1	101.3
PLife-1001	1.07	1.07	98.3	102.6
Nbody-251	1.08	1.10	96.4	100.5
Nbody-501	1.05	1.05	96.4	100.5
Nbody-1001	1.03	3.34	96.4	100.9
WSend-251	1.00	1.38	96.0	100.9
WSend-501	1.00	1.36	96.2	102.4
WSend-1001	1.00	1.27	96.0	103.8
WShift-251	1.01	1.02	96.2	100.3
WShift-501	1.00	1.01	96.2	100.3
WShift-1001	1.02	1.04	96.6	100.9
Random	125.1	136.80	523.3	637.0

Table 5.13: Comparison with offline offset-based representation

the space required to store the permutations. Since a significant number of permutations is generated for Random, the space required per event is more than for the offline case where only base timestamps are the major contributor to the space required for representation.

Similarly, for a number of applications the overall search is almost identical for the offline and online schemes. However, for WaveSend and NBody-1001, the average search is higher than the offline case. For WaveSend we believe this to be the result of the interval that is selected dynamically for representation. For a very small interval, e.g., 5 the overall search is very close to the offline case (shown in Table 5.10), however, for a small trace-reorder interval the number of permutations generated is rather large and results in significantly higher overall computational cost. For the NBody application with 1001 traces, all the online trace-reordering schemes result in relatively higher average search than the offline case and investigating this difference is part of the future work.

Overall, the results show that with little overhead we can generate very efficient partial-order representations in an online setting and by using the dynamic-interval trace-reordering scheme we can avoid the need for specifying offline configuration parameters for each application.

5.4.3 Storing Partial-Order Representation in a Database

In this section we look at the overhead associated with storing the offset-based representation in a relational database as opposed to a flat-file where each event is represented using a fixed number of bytes, i.e.,

$$Bytes/Event = 28 + 4 + 4 + (4 \times 16) = 100 \tag{5.4}$$

where 28 bytes are for event meta-data, two 4-byte pointers to a base timestamp and a permutation, and the space required for storing offsets. Any additional space required per event for the partial-order representation is a result of the space required for storing the base timestamps and permutations.

For our analysis, we consider the space required for storing the representation in MySQL 5.1 [47] with the MyISAM storage engine for tables. For other DBMS systems the space required would vary based on the exact format used for storing data. Furthermore, the space-requirement calculation is an approximation based on the information available about the storage requirements for different data types.

The base timestamps and permutations are stored in a similar format to a flat-file; each row contains a single vector element and a vector timestamp can be stored in N rows, where N is the number of traces. A reason for storing base timestamps and permutations this way is that it allows direct access to any element of the vector without any space overhead.

The space required for storing the base timestamps and permutations is therefore, the same as for the flat-file format, i.e., $4 \times N \times (B + P)$.

In a database, unlike a flat-file, only offsets that are used for representing an event are stored. The following row format is used for each event offset:

```
TRACE, INDX, BTS_INDX, PERM_INDX, FIRST_INDEX, LAST_INDEX, OFFSET, INCR
```

TRACE and INDX are the trace number and the event number of an event and serve as a unique identifier for an event. The FIRST_INDEX, LAST_INDEX, OFFSET and INCR make up a single offset. Each offset is stored in a separate row using the above representation. Note that the design is not completely normalized as the BTS_INDX and PERM_INDX are stored as many times as the number of offsets used by an event. A reason for this design is to provide speedier access to event offsets and permutation and base timestamp index, otherwise requiring a join operation. Finally, for events that do not have any offsets because a new base timestamp was generated for them, one row is stored with offset fields set to -1.

For the incremented-sequence offset scheme used exclusively for the online offset-based partial-order representation, the four fields representing an offset (FIRST_INDX, LAST_INDX, OFFSET, INCR) cannot be NULL. The events which require only one row for representation and are not counted towards the offsets include the first event on each of the N traces (are 0) and all other events which resulted in the creation of new base timestamp. Although there are other cases, such as for synchronous events where the same base time-stamp may be used to represent both events, we ignore such cases for approximation purposes. The size of each row according to MySQL documentation is given as follows [46]:

```
row length = 1 + (sum of column lengths)
```

```
+ (number of NULL columns + delete_flag + 7)/8
+ (number of variable-length columns)
```

To avoid expending extra space for NULL columns, all columns are set to be not NULL and a value of '-1' is used instead for events which require no offsets. The row size in bytes for event offsets is thus given by:

$$ROW\_SIZE = 1 + (8 \times 4) + 1 + 0 = 34\ Bytes \tag{5.5}$$

The total space without indexes can be approximated as follows:

$$PO\ Bytes \approx 4 \times N \times (B + P) + (NUM\_OFFSETS + N + B) \times ROW\_SIZE \tag{5.6}$$

Storing the representation in a relational database without creating suitable indexes would be of little use for applications that do require random access to an event's partial order representation. For different POET clients, the offsets are always accessed using the event identifier, i.e., TRACE and INDX, therefore, we use a joint index on the two columns. The worst-case size of the index file when all keys are inserted in a sorted order can be roughly approximated by $(KEY\_LENGTH + 4)/0.67$ for each key. The total space required for the partial-order representation with indexes is given by

$$Bytes \approx 4 \times N \times (B + P) + (NUM\_OFFSETS + N + B) \times ROW\_SIZE + E \times 17.91 \tag{5.7}$$

Table 5.14 shows the space required for storing the partial-order representation in MySQL using the MyISAM storage engine. Equations 5.6 and 5.7 are used to calculate the space required for storing the representation in a database without and with indexes. As noted before, when storing the representation in a database, it is of little value to store the representation without indexes. Furthermore, different monitoring and debugging

Application	Events	Avg. Offsets/Event	Storage Format		
			Flat File	DB	DB w/ Index
Life-250	502500	3.95	72.3	134.5	152.4
PLife-251	101502	2.67	72.7	91.7	109.6
PLife-501	203002	2.40	73.3	82.9	100.9
PLife-1001	406002	2.20	74.6	77.5	95.4
Nbody-251	1256502	3.95	72.5	134.9	152.8
Nbody-501	5013002	3.97	72.5	135.7	153.6
Nbody-1001	20026002	3.71	72.9	127.3	145.2
WSend-251	500248	3.89	72.9	133.2	151.1
WSend-501	1002498	3.88	74.4	134.2	152.1
WSend-1001	2006998	3.68	75.8	129.1	147.0
WShift-251	500248	3.96	72.3	134.8	152.7
WShift-501	1002498	3.49	72.3	118.9	136.8
WShift-1001	2006998	3.95	72.9	135.1	153.0
Random	53248	0.32	609.3	566.6	584.5

Table 5.14: Average bytes/event for storing partial order representation in database

applications may have different partial-order access patterns and may require additional indexes, which would contribute further to the total space required for representation. As shown in Table 5.14, with a single joint index the space required for representation is significantly more than the flat-file format. Although in a flat file we reserve space for all OFFSET_LIMIT offsets, whereas only the used offsets are stored in the database, the average number of offsets is very close to the OFFSET_LIMIT of 4. Therefore, the ability to save only the used offsets does not translate into lower space consumption for the partial-order representation. The above however, does not hold for Random, where significantly fewer events are represented using offsets and we see space saving when the representation is stored in database.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Event partial orders are fundamental to monitoring and debugging distributed and parallel applications. The existing techniques for representing event partial orders are not scalable for large applications with hundreds or thousands of processes. Furthermore, the techniques that do show some promise in conserving space are static in nature and cannot be used in a dynamic setting, prohibiting online monitoring and debugging. Although, efficiently constructing partial orders is an active area of research in the distributed-systems and database communities, the end goals are different and by taking advantage of the inherent structure of distributed applications, more efficient event partial orders can be constructed.

In this work, we adapt the efficient offset-based partial order representation schemes proposed by Taylor [61] to an online setting. We develop a layered client architecture and build the proposed online trace-reordering scheme directly into POET. We propose a completely dynamic trace-reordering scheme which is used in conjunction with the offset-

based schemes to construct scalable event partial orders in real-time. The significant conclusions of our work are as follows:

1. The correct trace order can result in significant space and computational efficiency when using offset-based event-partial-order representations. This is a re-affirmation of the offline results already known [61].

2. Unlike base timestamps, only the most recent order of traces is likely to be useful for representing new events using offsets. This helps in significantly pruning the combined search space when looking for a suitable base-timestamp and permutation combination.

3. The ideal trace reorder interval for generating an efficient representation for different distributed and parallel applications varies from application to application and depends on the intrinsic communication structure of the target application.

4. A dynamic trace-reordering scheme can yield good trace-reorder intervals for a broad set of applications, independent of the communication pattern exhibited by the application. Furthermore, the dynamic approach sacrifices little in terms of the computational efficiency to achieve space savings on a par with the offline offset-based scheme. This capability can be directly leveraged by tools such as POET for facilitating real-time monitoring and debugging of large distributed and parallel applications.

This work shows that online efficient partial order representations can be constructed for large applications with hundreds or thousands of processes. Furthermore, offset-based representation in conjunction with online trace reordering schemes can be leveraged effectively by various tools to monitor and debug large distributed and parallel applications,

which previously ran into severe scalability issues due to the size of partial order representations.

6.2 Future Work

There are a number of possible optimizations that can be made to the dynamic trace-reordering scheme. A simple such optimization is to check for the same permutation being generated repeatedly. If such a repetition does occur, it would indicate that the traces are being reordered more often than required by the application. Increasing the trace-reorder interval in such a case may be helpful in further reducing the total number of permutations. In our current implementation, the complete traffic matrix is used for reordering traces. The motivation for the complete traffic matrix came from the effectiveness of the offline offset-based schemes which only utilized a single trace order. The above also shows that distributed applications do exhibit a global communication pattern. As described in Chapter 5, most distributed applications undergo periods of flux where the communication pattern changes significantly. Leveraging only the most recent communication pattern may prove to be useful in producing a trace order that works during such phases.

Partial-order-event-data access patterns vary for different monitoring and debugging operations. Effectively leveraging the storage and caching of offset-based representations to provide efficient access to partial-order event data is another area of future work. Lastly, evaluating the effectiveness of the online trace-reordering scheme for an even larger set of applications is part of the future work and would be significantly useful in gaining valuable insights for further improving the efficiency of the offset-based partial-order representation schemes.

References

[1] Amazon Elastic Compute Cloud and Amazon Relational Database Service disruption. http://aws.amazon.com/message/65648/. 3

[2] Amazon's Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/. 2, 3

[3] Amazon's Simple Storage Service (S3). http://aws.amazon.com/s3/. 2

[4] Amazon's SimpleDB. http://aws.amazon.com/simpledb/. 2

[5] Apache Cassandra project. http://cassandra.apache.org. 2

[6] Heroku cloud application platform. http://www.heroku.com/. 2

[7] Microsoft Azure. http://www.microsoft.com/windowsazure/. 2

[8] P. Almeida, C. Baquero, and V. Fonte. Interval tree clocks. *Principles of Distributed Systems*, pages 259–274, 2008. 29

[9] E. Anderson, M. Arlitt, C.B. Morrey III, and A. Veitch. Dataseries: An efficient, flexible data format for structured serial data. *ACM SIGOPS Operating Systems Review*, 43(1):70–75, 2009. 23

[10] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. 2

[11] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, and S.W. Williams. The landscape of parallel computing research: A view from Berkeley. Technical report, 2006. 2

[12] P.C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, 1995. 3

[13] A. Bialecki, M. Cafarella, D. Cutting, and O. OMalley. Hadoop: A framework for running applications on large clusters built of commodity hardware. *http://lucene.apache.org/hadoop*, 2005. 2

[14] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996. 3

[15] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008. 2

[16] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, 1991. 34

[17] W.H. Cheung. *Process and event abstraction for debugging distributed programs*. PhD thesis, 1989. 21

[18] J.D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of 15th*

International Symposium on Parallel and Distributed Processing, pages 10–pp. IEEE, 2001. 5

[19] M. Christiaens and K. De Bosschere. Accordion clocks: Logical clocks for data race detection. *Euro-Par 2001 Parallel Processing*, pages 494–503, 2001. 28

[20] M.P. Consens, M. Hasan, and A.O. Mendelzon. Debugging distributed programs by visualizing and querying event traces. In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 181–183, 1993. 3

[21] Jeff Dean. Designs, lessons and advice from building large distributed systems. *http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf*, 2010. 2

[22] O. Ekpenyong. Parallel pattern search in large, partial-order data sets on multi-core systems. Master's thesis, University of Waterloo, 2011. 5

[23] D.J. Ernst and D.E. Stevenson. Concurrent CS: Preparing students for a multicore world. In *ACM SIGCSE Bulletin*, volume 40, pages 230–234. ACM, 2008. 2

[24] C. Fidge. Fundamentals of distributed system observation. *Software, IEEE*, 13(6):77–83, 1996. 4, 6, 17

[25] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proceedings of 10th International Conference on Distributed Computing Systems*, pages 134–141. IEEE, 1990. 17

[26] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M. Woodside. Performance analysis of distributed server systems. *Carleton University, Ottawa, Ont., Canada*, 2000. 3

[27] M. Frumkin, R. Hood, and J. Yan. On the information content of program traces. 1998. 28

[28] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005. 2

[29] A. Geist, J. Kohl, and P. Papadopoulos. Visualization, debugging, and performance in PVM. In *Proceedings of Visualization and Debugging Workshop*, 1994. 39

[30] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003. 2

[31] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic monitoring of high-performance distributed applications. In *Preceedings of 11th IEEE International Symposium on High Performance Distributed Computing*, pages 163–170. IEEE, 2002. 3

[32] D. Haban and D. Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *Software Engineering, IEEE Transactions on*, 16(2):197–211, 1990. 3

[33] M.T. Heath. Recent developments and case studies in performance visualization using paragraph. In *Proceedings of the Workshop on Performance Measurement and Visualization of Parallel Systems*, pages 175–200. Elsevier Science Publishers BV, 1993. 39

[34] High Performance Computing Center Stuttgart. *Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Version 2.1.*, 2008. 62

[35] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M.M. Guzmán, K. Hammond, J. Hughes, and T. Johnsson. Report on the programming language

Haskell: A non-strict, purely functional language version 1.2. *ACM Sigplan Notices*, 27(5):1–164, 1992. 2

[36] C. Jard and G.V. Jourdan. Dependency tracking and filtering in distributed computations. *Research Report, Institut de recherche en informatique et systèmes aléatoires*, 851. 17

[37] G. Jiang, H. Chen, and K. Yoshihira. Profiling services for resource optimization and capacity planning in distributed systems. *Cluster Computing*, 11(4):313–329, 2008. 3

[38] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Transactions on Database Systems (TODS)*, 36(1):7, 2011. 31, 33

[39] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: A high-compression indexing scheme for reachability query. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 813–826. ACM, 2009. 31

[40] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *Journal of Computer and System Sciences*, 65(1):150–167, 2002. 31, 32

[41] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, (4):424–436, 1985. 3

[42] T. Kunz, J.P. Black, D.J. Taylor, and T. Basten. POET: Target-system independent visualizations of complex distributed-application executions. *The Computer Journal*, 40(8):499, 1997. 3, 7, 40

[43] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. 15

[44] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989. 6, 17

[45] Frank Miguel. Monitoring and response for distributed systems. *http://www.ibm.com/developerworks/webservices/library/wa-monresp/*, 2004. 1

[46] MySQL. Table column-count and row-size limits, http://dev.mysql.com/doc/refman/5.0/en/column-count-limit.html, 2011. 99

[47] AB MySQL. MySQL: The world's most popular open source database, http://www.mysql.com/, 2005. 98

[48] Simon Neville. Google's email service wipes entire email accounts of 150,000 gmail users. *http://www.dailymail.co.uk/sciencetech/article-1361334/Googles-email-service-wipes-entire-accounts-150-000-Gmail-users.html*, 2011. 3

[49] M. Nichols. *Efficient pattern search in large, partial-order data sets.* PhD thesis, University of Waterloo, 2008. 5

[50] M. Odersky et al. The Scala programming language. *http://www. scala-lang. org*, 2008. 2

[51] O. Ore. *Theory of graphs*, volume 38. Amer Mathematical Society, 1967. 17, 35

[52] M. Raynal and M. Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996. 30

[53] G.G. Richard III. Efficient vector time with dynamic process creation and termination. *Journal of Parallel and Distributed Computing*, 55(1):109–120, 1998.

[54] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, pages 184–191. ACM, 2004. 31

[55] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152, 1999. 5

[56] C.E. Shannon and W. Weaver. *The mathematical theory of communication*. 1959. 28

[57] M.B. Sheikh, U.F. Minhas, O.Z. Khan, A. Aboulnaga, P. Poupart, and D.J. Taylor. A Bayesian approach to online performance modeling for database appliances using gaussian models. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, pages 121–130. ACM, 2011. 3

[58] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *IEEE's 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010. 2

[59] J.A. Summers. Precedence-preserving abstraction for distributed debugging. Master's thesis, University of Waterloo, 1992. 17, 37

[60] D. Taylor. Scrolling partially ordered event displays. *Journal of Parallel and Distributed Computing*, 65(5):643–653, 2005. 40

[61] David Taylor. Efficient representation of event partial orders. 2009. 5, 6, 7, 17, 30, 40, 43, 44, 49, 51, 53, 54, 55, 59, 62, 63, 65, 67, 68, 72, 75, 76, 93, 96, 102, 103

[62] D.J. Taylor. Event displays for debugging and managing distributed systems. In *Proceedings of the International Workshop on Network and Systems Management*, pages 112–124, 1995. 5

[63] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009. 2

[64] W.T. Trotter. *Combinatorics and partially ordered sets: Dimension theory*, volume 6. Johns Hopkins Univ Pr, 2001.

[65] X. Wang, J. Mayo, W. Gao, and J. Slusser. An efficient implementation of vector clocks in dynamic systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 593–599, 2006. 30

[66] P. Ward and D. Taylor. Self-organizing hierarchical cluster timestamps. *Euro-Par 2001 Parallel Processing*, pages 46–56, 2001. 17

[67] P.A.S. Ward. An online algorithm for dimension-bound analysis. *Euro-Par99 Parallel Processing*, pages 144–153, 1999. 34

[68] P.A.S. Ward. A framework algorithm for dynamic, centralized dimension-bounded timestamps. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative Research*, page 14. IBM Press, 2000. 17, 34

[69] P.A.S. Ward. *A Scalable Partial-order data structure for distributed-system observation.* PhD thesis, University of Waterloo, 2001. 18, 25

[70] P.A.S. Ward, T. Huang, and D.J. Taylor. Clustering strategies for cluster timestamps. 2004. 37, 39, 44, 72

[71] P.A.S. Ward and D.J. Taylor. A hierarchical cluster algorithm for dynamic, centralized timestamps. In *21st International Conference on Distributed Computing Systems*, pages 585–593. IEEE, 2001. 17, 37, 38, 72

[72] S. Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962. 13

[73] M. Yannakakis. The complexity of the partial order dimension problem. *SIAM Journal on Algebraic and Discrete Methods*, 3:351, 1982. 36

[74] H. Yildirim, V. Chaoji, and M.J. Zaki. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284, 2010. 31, 32, 33